STRUCTURED COBOL

A. S. PHILIPPAKIS
LEONARD J. KAZMIER

THIRD EDITION

STRUCTURED COBOL

THIRD EDITION

A. S. PHILIPPAKIS
Arizona State University

LEONARD J. KAZMIER
Arizona State University

McGRAW-HILL BOOK COMPANY
New York St. Louis San Francisco Auckland Bogotá Hamburg
Johannesburg London Madrid Mexico Montreal New Delhi
Panama Paris São Paulo Singapore Sydney Tokyo Toronto

STRUCTURED COBOL

1 2 3 4 5 6 7 8 9 0 WEBWEB 8 9 8 7 6 5

ISBN 0-07-049809-1

This book was set in Times Roman by Monotype Composition Company, Inc.
The editors were Christina Mediate and Joseph F. Murphy;
the designer was Rafael Hernandez;
the production supervisor was Marietta Breitwieser.
The drawings were done by Accurate Art, Inc.
The cover photograph was taken by R. Wickham.
Webcrafters, Inc., was printer and binder.

See Acknowledgment on page xv. Copyrights included on this page by reference.

Library of Congress Cataloging-in-Publication Data

Philippakis, Andreas S.
Structured COBOL.

Includes index.
1. COBOL (Computer program language) 2. Business—Data processing. I. Kazmier, Leonard J. II. Title.
HF5548.5.C2P54 1986 005.13'3 85-17083
ISBN 0-07-049809-1

TO OUR PARENTS

CONTENTS

PREFACE

COBOL (COmmon Business Oriented Language) has become the standard programming language for administrative applications of the computer in industry and government. The language has been continuously reviewed and modified under the sponsorship of the Conference on Data Systems Language (CODASYL), a working committee composed of representatives from major computer language users and computer manufacturers. The revision of COBOL currently being implemented replaces the 1974 version, which was itself a refinement of earlier versions of the language.

A special feature of this third edition is the inclusion of new language features and enhancements that are now available with the revised version of COBOL. The description of these enhancements is highlighted in color throughout the book. Therefore it is easy to distinguish what language features can be used with compilers based on the 1974 standard as contrasted to the new version of COBOL.

This book has been designed to facilitate the learning of COBOL. It includes thorough coverage of concepts, as well as examples of applications, at both elementary and intermediate levels of complexity. Review sections are included in the text as opportunities for the reader to enhance effective learning through self-testing. Students in business administration and computer science will find this book to be a suitable guide to the language.

The concepts of *structured programming* are used throughout this book. A structured program is one which follows certain principles of modular design and results in simplification of programming tasks and improved self-documentation. COBOL is particularly well suited for the application of structured programming concepts, and use of these concepts has been incorporated in all of the computer programs that are described.

The book consists of fifteen chapters. The first two chapters present a complete set of elementary language statements so that the student can begin to write

complete programs in a relatively short time. Beyond the introductory material in the first two chapters, the language concepts and options are developed at two levels. Chapters 4 through 6 present a comprehensive set of language statements, including conditional statements. Chapters 8 through 15 develop special language features as they relate to such areas as file processing, table handling, and subprograms. Chapters 3 and 7 are concept-oriented chapters concerned with program structure and program design, respectively. Several chapters have also been restructured to improve the presentation and to enable students to write complete COBOL programs sooner.

The authors express their appreciation to Christina Mediate for her capable supervision of this project in her role as editor. We also extend thanks to several of the following reviewers for their comments and recommendations with respect to this revision: John Beidler, University of Scranton; Frank Cable, Pennsylvania State University; John Hirschbuhl, Computer Knowledge International; Leo Niemi, Western Michigan University; and Michael Weinstein, California State University.

A. S. Philippakis
Leonard J. Kazmier

ACKNOWLEDGMENT

The following acknowledgment is reprinted from *American National Standard Programming Language COBOL, X3.23-1974* published by the American National Standards Institute, Inc.

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention 'COBOL' in acknowledgment of the source, but need not quote the acknowledgment):

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell.

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

and the introductory material [illegible] the language features and options are developed at two levels. Chapters 4 through 8 present a comprehensive set of language statements, including conditional statements. Chapters 8 through 12 develop special language features as they relate to such areas as file processing, table handling, and subprograms. Chapters [illegible] and [illegible] are concept-oriented chapters concerned with program structure and program design, respectively. Several chapters have also been restructured to improve the presentation and to enable students to write complete COBOL programs sooner.

The authors express their appreciation to Christina Mediate for her capable assistance in this project in her role as editor. We also extend thanks to several of the following reviewers for their comments and recommendations with respect to this revision: John Bradley, University of Scranton; Frank Cable, Pennsylvania State University; Jack Hirschfeld, Computer Knowledge International; Leo [illegible], Western Michigan University; and Michael Weinstein, California State University.

A. S. Philippakis
Leonard J. Kazmier

The following acknowledgment is reprinted from American National Standard Programming Language COBOL, X3.23-1974 published by the American National Standards Institute, Inc.

Any organization interested in reproducing the COBOL standard and specifications in whole or in part, using ideas from this document as the basis for an instruction manual or for any other purpose, is free to do so. However, all such organizations are requested to reproduce the following acknowledgment paragraphs in their entirety as part of the preface to any such publication (any organization using a short passage from this document, such as in a book review, is requested to mention "COBOL" in acknowledgment of the source, but need not quote the acknowledgment).

COBOL is an industry language and is not the property of any company or group of companies, or of any organization or group of organizations.

No warranty, expressed or implied, is made by any contributor or by the CODASYL Programming Language Committee as to the accuracy and functioning of the programming system and language. Moreover, no responsibility is assumed by any contributor, or by the committee, in connection therewith.

The authors and copyright holders of the copyrighted material used herein

FLOW-MATIC (trademark of Sperry Rand Corporation), Programming for the UNIVAC I and II, Data Automation Systems copyrighted 1958, 1959, by Sperry Rand Corporation; IBM Commercial Translator Form No. F28-8013, copyrighted 1959 by IBM; FACT, DSI 27A5260-2760, copyrighted 1960 by Minneapolis-Honeywell

have specifically authorized the use of this material in whole or in part, in the COBOL specifications. Such authorization extends to the reproduction and use of COBOL specifications in programming manuals or similar publications.

STRUCTURED COBOL

1 OVERVIEW OF COBOL PROGRAMMING

INTRODUCTION

Administrative data processing systems supply managers with information needed for decision making by providing for efficient data collection, data processing, and production of required reports. Data input typically is voluminous and includes both numeric and nonnumeric data records. For instance, sales data would include purchase amounts and customer names. Further, the processing of business data typically involves file updating. Input data are processed with respect to historical files, such as customer accounts, in order to update the data in the files. The output of administrative data processing is characterized by the production of reports that group and summarize data by meaningful categories that correspond to various organizational functions, such as marketing-oriented reports and financially oriented reports.

In contrast to administrative data processing, scientific computing generally involves a lower volume and diversity of input data, small or nonexistent files, less complex processing logic but more extensive mathematical manipulation, and more limited report production needs. Because administrative data processing has characteristics different from those of scientific computing, a special programming language has been developed to fulfill the particular needs associated with such processing of data. The language is COBOL (COmmon Business Oriented Language). Two other languages also used for data processing applications, particularly with small computers, are BASIC (Beginner's All-purpose Symbolic Instruction Code) and RPG II (Report Program Generator II). However, COBOL has persisted as the most widely used language for administrative data processing.

In this chapter we introduce the structure of COBOL programs, describe how data-names are differentiated from data content, and explain how both numeric

and nonnumeric constants are coded. These associated topics then serve as the cornerstone for discussion of a sample COBOL program concerned with producing a sales report that lists units sold according to customer name. The final sections of the chapter are concerned with the COBOL Coding Form and the operating system of the computer as it relates to the execution of COBOL programs.

LEVELS OF COMPUTER LANGUAGES

A computer program is a set of instructions that directs a computer in the performance of a data processing task. A computer language is a set of characters, words, and rules that can be used to write a computer program.

Every computer model has its own language, which is determined by its hardware structure. Such "native" computer languages are referred to as *machine* languages. These languages are, of course, machine dependent and are, at a first glance, highly obscure because they consist of long strings of numeric codes. Early computer programming was almost exclusively machine-language programming. Although machine language is natural to the hardware of a computer, it is quite unnatural to human programmers. A step in the direction of facilitating programming was taken with the development of *symbolic* languages. Symbolic languages use mnemonic codes, such as ADD, to represent machine instructions. For instance, a machine instruction such as 21300400, meaning to add the value stored in location 300 to location 400, could be written, ADD Amount 1, Amount 2. Mnemonic codes, of course, are not understood by a computer; therefore, they have to be translated into machine-language form. An *assembler* is a machine-language program that translates symbolic language instructions into machine-language instructions. Symbolic languages are machine dependent in that a set of mnemonic codes is applicable only to a particular computer model; therefore, the programmer has to be familiar with the particular instruction repertoire of the machine being programmed. This is a serious disadvantage when programming efforts are extensive and hardware is changing.

The next stage in the development of programming language was the advent of *higher-level* languages that are procedure oriented rather than machine oriented. Such languages focus on the data processing procedure to be accomplished rather than on the coding requirements of particular machines. Further, higher-level languages are *not* machine dependent: such programs are not restricted to use with particular computer models. Even though such instructions are not designed to correspond to the way a particular computer model operates, they of course ultimately must be executed on some particular machine. Again, the process of translation is used to obtain the required machine-language program. A computer program written in a higher-level language is referred to as a *source* program. A *compiler* is a machine-language program that translates (or compiles) the source program into a machine-language program, which is referred to as the *object* program. The object program is then input into the machine to perform the required task. Thus, a compiler is a program whose function it is to convert source programs into object programs. The main difference between a compiler and an assembler is that *compilation* is a more complex process than *assembly*. Assembly typically involves a one-for-one translation from a mnemonic to a machine code; compilation involves a many-for-one translation. One higher-level instruction may be the equivalent of several machine-level instructions.

Figure 1-1 illustrates the compilation and execution process. In the first phase, the source program and the compiler serve as input. The output includes the object program, which is stored on a magnetic disk or tape device, and a listing of the source program on the printer, along with diagnostic error messages. If no

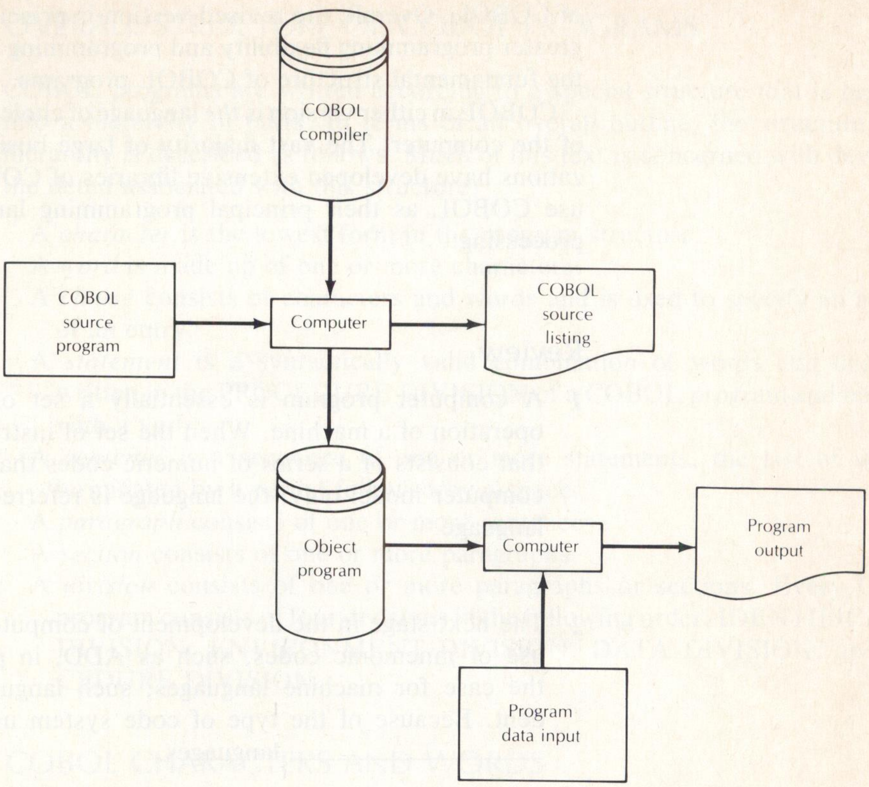


FIGURE 1-1
COBOL PROGRAM COMPILATION-EXECUTION PROCESS.

serious errors are detected, the object program is entered into the computer. Based on the program instructions, input data is then read and analyzed, culminating with the required output.

COBOL is a programming language that has been designed expressly for administrative data processing. It is a higher-level language and, as such, generally is machine independent.

The idea of developing the language was conceived at a Pentagon meeting in May 1959. At that meeting, representatives from the government, from business users, and from computer manufacturers decided that it was feasible to proceed with the development of a higher-level language that would satisfy the specific needs of administrative data processing, as contrasted to scientific computing. A preliminary version of COBOL appeared in December 1959. This version was followed in 1961 by COBOL-61, which became the cornerstone for the development of later versions of the language. In 1968 a standard version of the language was approved by what now is called the American National Standards Institute (ANSI), and a revised version was adopted by ANSI in 1974. COBOL is continuously evolving and being enhanced by the addition of new programming capabilities and the deletion of obsolete functions. Proposed changes in the language are documented annually in the *Journal of Development*. These changes are tentative, however, until ANSI adopts a new standard.

Following several years of study and discussion regarding the ways in which COBOL-74 could be improved, a new standard is expected to be adopted sometime in late 1985. Because of the large number of existing programs written with respect to the 1974 standard, it will take some time before the features associated with the most recent revision are widely used. Therefore, in this text we make a clear distinction between the 1974 standard and the revised version

of COBOL. Overall, the revised version is principally concerned with achieving greater programming flexibility and programming capability, and does not change the fundamental structure of COBOL programs.

COBOL in either version is *the* language of choice for administrative applications of the computer. The vast majority of large business and governmental organizations have developed extensive libraries of COBOL programs and continue to use COBOL as their principal programming language for administrative data processing.

Review

1 A computer program is essentially a set of instructions that directs the operation of a machine. When the set of instructions is written in a language that consists of a series of numeric codes that can be used with a particular computer model only, the language is referred to as a ________________ language.

machine

2 The next stage in the development of computer languages made possible the use of mnemonic codes, such as ADD, in place of numeric codes. As is the case for machine languages, such languages are also machine dependent. Because of the type of code system used, such languages are called ________________ languages.

symbolic

3 The third stage in the development of computer languages was the formulation of higher-level, or procedure-oriented, languages that [are also / are not] machine dependent. An example of such a language is ________________.

are not; COBOL (*or* FORTRAN, BASIC, PASCAL, PL/I, *etc.*)

4 Both symbolic languages and procedure-oriented languages have to be translated into machine-language form before they can be used to direct computer operations. The program that translates a symbolic language program into machine-language form is called a(n) ________________, whereas the program that translates a procedure-oriented language program is called a(n) ________________.

assembler; compiler

5 In the context of using a procedure oriented language, the program written in such a language is often referred to as the ________________ program, and the translated version of the program is referred to as the ________________ program.

source; object

6 The procedure-oriented language that has been designed specifically to satisfy programming needs associated with administrative data processing is ________________.

COBOL

7 Development of the COBOL language is monitored, and new versions of the language are approved periodically, by the ________________ ________________.

American National Standards Institute (ANSI)

OVERALL STRUCTURE OF COBOL PROGRAMS

COBOL programs are written according to a special structure that is organized into a hierarchy of parts. In terms of an overall outline, the structure of this hierarchy is described as follows. Much of this text is concerned with developing the detail associated with this structure.

A *character* is the lowest form in the program structure.

A *word* is made up of one or more characters.

A *clause* consists of characters and words and is used to specify an attribute of an entry.

A *statement* is a syntactically valid combination of words and characters written in the PROCEDURE DIVISION of a COBOL program and beginning with a verb.

A *sentence* is a sequence of one or more statements, the last of which is terminated by a period followed by a space.

A *paragraph* consists of one or more sentences.

A *section* consists of one or more paragraphs.

A *division* consists of one or more paragraphs or sections. Every COBOL program consists of four divisions in the following order: IDENTIFICATION DIVISION, ENVIRONMENT DIVISION, DATA DIVISION, and PROCEDURE DIVISION.

COBOL CHARACTERS AND WORDS

The most basic and indivisible unit of the COBOL language is the *character*. The set of characters used to form COBOL source programs consists of the 51 characters identified in Figure 1-2.

A sequence of characters can form a *word*. There are two types of COBOL words: *reserved words* and *user-defined words*. Reserved words are defined by the COBOL language. (Appendix A presents a complete list of COBOL reserved words.) The programmer cannot use any of these words except in the form specified by the language. Much of this text is devoted to explaining the use of such reserved words.

User-defined words are supplied by the programmer (language user) in order to satisfy the format of a clause or statement in the language. A user-defined word may be 1 to 30 characters in length and may consist of letters, digits, and hyphens, except that a hyphen may not appear as the first or last character.

There are 17 types of user-defined words, examples of which are: condition-name, data-name, paragraph-name, record-name, and file-name. The reader will be exposed to different types of user-defined words as the text progresses. Special attention is given to data-names in the section that follows, because data-names are used so frequently.

CHARACTER	MEANING
0, 1, . . . , 9	Digit
A, B . . . Z	Letter
	Space (blank)
( ) . , "	Special symbols
+ − * /	
= $;	
< >	

FIGURE 1-2
THE SET OF CHARACTERS IN COBOL PROGRAMMING.

Review

1 The basic units of the COBOL language, such as the numbers 0 through 9 and the capital letters of the alphabet, are referred to as ________.

characters

2 COBOL words specifically defined by the COBOL language and used only for particular purposes are called ________ words.

reserved

3 COBOL words that are formulated by applications programmers are called ________ words.

user-defined

DATA-NAMES

Central storage or main storage in a computer can be thought of as a long string of character positions, with direct access available to any position. Both data and instructions are stored in internal storage, but, from the programmer's viewpoint, data storage is the most important concern. Data are stored in certain positions such that they can be referred to by a name or an address. In COBOL, the addresses are symbolic names and are called *data-names*. One way to view a computer program is to say that it consists of a set of instructions to manipulate central storage areas that are referenced by their corresponding data-names. The idea will become clear as we proceed.

Data-names are coined at the discretion of the programmer, except that there are certain rules that must be followed.

1 A data-name can be up to 30 characters in length and can include alphabetic characters, numeric characters, and hyphens.
2 At least one character must be alphabetic.
3 The only special symbol permitted is the hyphen. A hyphen must always be embedded; that is, it cannot be the first or last character of the data-name.
4 Blanks cannot be included in the data-names.
5 Within the above rules the programmer may use any data-name, with the exception of the approximately 300 COBOL reserved words listed in Appendix A. (Manufacturers often add some of their own words to the ANSI list.)

Some examples of legitimate data-names are:

HOURS	PREMIUM
ENDING-INVENTORY	A527157
SALES-TAX-TOTAL	31576X5

Of course, data-names do not have to be meaningful English words. A programmer can choose to use such data-names as X, Y, Z, X1, X2, and the like. However, even though such data-names are typically shorter than those that are inherently meaningful as names, they increase the likelihood of subsequent confusion. COBOL was designed specifically to allow self-documentation, which means that by reading the program one should be able to understand what the program does and what data it uses. The problem with using cryptic data-names is that their meanings are forgotten by the programmer and are never understood by others unless a list of definitions is supplied.

Part of the programming job is to subdivide central storage into data units, such as characters, elementary or group items, and records. This is accomplished by using data-names in ways that can reflect the structure of the data involved.

(Data name)	CUSTOMER-ADDRESS							
(Data name)	NAME		STREET		CITY-STATE		ZIP-CODE	
(Data name)	FIRST-NAME	LAST-NAME	STREET-NUMBER	STREET-NAME	CITY	STATE	PO	ZONE
(Content)	RONALD	JOHNSON	1057	MONTEREY DRIVE	TEMPE	ARIZONA	852	82
	10	22	27	42	52	62	65	67

FIGURE 1-3
CONCEPTUAL STRUCTURE OF INFORMATION IN INTERNAL STORAGE.

As a matter of fact, COBOL derives much of its suitability for administrative applications from the opportunity it provides to take explicit account of the structure of data.

As an example of the use of data-names, consider the following information that is to be stored:

RONALD JOHNSON
1057 MONTEREY DRIVE
TEMPE, ARIZONA 85282

Internally, the information will be stored as a continuous sequence of characters rather than as three lines. Assuming that it will be stored starting with storage position 1, we have the conceptual structure presented in Figure 1-3. The bottom row in the figure identifies the data as well as the data positions in internal storage, and the other rows refer to the labels by which reference can be made to the storage locations that contain particular kinds of information. In other words, in this figure we have defined storage fields and labeled them by the use of different data-names. For example, the label FIRST-NAME refers to storage positions 1–10. Correct computer programming relies on the use of unique data-names to identify unique positions in storage. Thus, we can write instructions that direct the computer to take the data in columns 1–10 of an input record and enter it in the storage location called FIRST-NAME. Obviously, the data-name FIRST-NAME must be unique to avoid ambiguity. Similarly, we could instruct the machine to print the contents of STREET on the printer, which would thereby make reference to the information held in storage positions 23–42.

Remember that a label or data-name, such as STREET, does not have any conceptual meaning for the computer, as it would have for a person. We could just as well have used the data-name XYZ in our example. We simply indicate to the machine that something is to be done to or with the content of a storage location that has been labeled for reference purposes. Even if the label implies something about the meaning of the content to the programmer, it does not imply anything like that to the machine. Of course, it is a good habit to coin labels that convey meaning about the content to the programmer.

Referring again to Figure 1-3, notice that FIRST-NAME refers to storage positions 1–10, NAME refers to storage positions 1–22, and CUSTOMER-ADDRESS refers to storage positions 1–67, which include all the positions of this example. The way we have used the data-names allows us to reference data in accordance with a desired structure. We can refer to the whole record, group items in the record, or to elementary items.

Review

1 IN COBOL, a label for a field of data appropriately is called a(n) ________ ________.

data-name

2 A data-name must not be more than ________________ (number) characters in length and can include [alphabetic characters only / alphabetic and numeric characters].

30; alphabetic and numeric characters

3 Every data-name must include at least one [alphabetic / numeric] character, and the only special symbol permitted is the ________________.

alphabetic; hyphen

4 Place a check mark before each of the following that is a legitimate data-name in COBOL.

a ________ INVENTORY
b ________ END OF YEAR BALANCE
c ________ 2735B5
d ________ 27-35B5
e ________ BALANCE-DUE-
f ________ 57
g ________ END-OF-YEAR-BALANCE-DUE-ON-ACCOUNT
h ________ DATA
i ________ BALANCE

a ✓
b *spaces not allowed*
c ✓
d ✓
e *hyphens must be embedded*
f *at least one alphabetic character must be included*
g *must be less than 30 characters in length*
h *a reserved COBOL word (a bit of a trick question—see Appendix A)*
i ✓

5 Each field of internal storage has a unique data-name, or label, associated with it. Each data-name [must / need not] be unique and [must / need not] imply something about the meaning of the content.

must; need not

6 Data-names can be so structured that data can be referenced according to a hierarchical organization. When this is done, the labeled data fields will [all be separate and distinct / involve some overlap].

involve some overlap (e.g., "STREET" vs. "STREET-NUMBER" in Figure 1-3

CONSTANTS

It may have occurred to you that a data-name in COBOL is analogous to the concept of a variable in algebra. It is a general symbol or name that can have many possible values. In addition to data-names, COBOL uses constants, and they are of three types: numeric literals, figurative constants, and nonnumeric

literals. As an example of a *numeric literal,* suppose that the sales tax rate in a particular state is 0.04 of sales. Therefore, within the COBOL program we need a way in which we can multiply the amount of sales by 0.04. One way of accomplishing this *without* using a numeric literal is to define a storage field, assign to it a data-name, such as TAX-RATE, and input the value 0.04 in the field. Conceptually, the internal storage location has the following structure:

TAX-RATE			(Data-name)
0	0	4	(Content)

The decimal point is not shown but is understood to be located in the appropriate position. With this approach, the reference to TAX-RATE will make available the 0.04 value stored in this field. Another option available in COBOL, however, is simply to write the numeric literal 0.04 in the program itself and use this value directly. Essentially, we are saying that, if a value is a numeric constant, its data-name in internal storage is the same as its value. Conceptually, the internal storage location has the following structure:

004			(Data-name)
0	0	4	(Content)

Numeric literals without a decimal point are understood to be integers (whole numbers). If a decimal point is used, it must not be the last character. Thus, 35. is not correct, whereas 35.0 is acceptable. The reason for this rule is that in COBOL programming, the period is always used to signal the end of a sentence, just as in English, and it would be ambiguous whether a point following a number is a decimal point or a period.

The second type of constant used in COBOL is the *figurative constant.* The most common figurative constants are ZERO, ZEROS, ZEROES, SPACE, and SPACES, although a few others are available. These refer to zeros or blanks, respectively. Their general use can be illustrated by the following brief examples. Suppose we want to set AMOUNT equal to zero. We can write MOVE ZERO TO AMOUNT to accomplish this objective. Similarly, if we wish to ascertain that blanks are contained in the field called TITLE, we can write MOVE SPACES TO TITLE. Figure 1-4 lists and defines the standard figurative constants used in COBOL programs.

In addition to numeric literals and figurative constants, the third class of constants is the *nonnumeric literal.* As contrasted to numeric literals and figurative constants, the nonnumeric literal is any alphanumeric value enclosed in quotation marks. For example, suppose we want to print the title INCOME STATEMENT. The words INCOME and STATEMENT are not intended to refer to data-names; rather, we simply want these exact words printed. This can be done by enclosing them in quotation marks and using them as illustrated in the next chapter, in which the PROCEDURE DIVISION of COBOL will be described.

As an example of how a nonnumeric literal might be used in a decision context, suppose we want to know if a customer's last name is BROWN. We could write something like this:

```
IF LAST-NAME EQUAL 'BROWN' (etc.)
```

Any letter, number, or special symbol can be enclosed in quotation marks with the exception of a quotation mark itself. If we want to use quotation marks as part of the literal, we can accomplish this by use of the QUOTE figurative constant as follows:

```
QUOTE 'TOTAL AMOUNT' QUOTE
```

ZERO ZEROS ZEROES	All three forms are equivalent, and they reference the value of zero. In an instruction such as MOVE ZEROS TO AMOUNT, the storage field AMOUNT would be filled by as many zeroes as there are positions in that field. Thus the context determines the number of occurrences of the character 0.
SPACE SPACES	Both forms are equivalent, and they reference one or more blanks, similar to the ZERO constant.
QUOTE QUOTES	Both forms are equivalent, and they reference the quotation mark.
HIGH-VALUE HIGH-VALUES	Both forms are equivalent, and they reference the highest value in the collating sequence for the particular computer system.
LOW-VALUE LOW-VALUES	Both forms are equivalent, and they reference the lowest value in the collating sequence for the particular computer system.
ALL literal	References one or more occurrences of the single character nonnumeric literal, as in MOVE ALL "A" TO HEADER, which results in the storage field HEADER being filled with A's.

FIGURE 1-4
FIGURATIVE CONSTANTS AND THEIR MEANINGS.

In this example the word QUOTE indicates a quotation mark. The use of the figurative constant QUOTE before and after the nonnumeric literal will result in quotation marks being printed. The quotation marks are printed as single or double marks, depending on the specific printer used.

Unlike a data-name, a nonnumeric literal can include blanks. The nonnumeric literal also can be composed entirely of numeric characters. This may seem like a contradiction, but it is not, since the term "nonnumeric" refers to how the characters are handled within the computer and not to their alphanumeric form as such.

Review

1 In all, three classes of constants were discussed in this section. They are the ________________, ________________, and ________________.

numeric literal; figurative constant; nonnumeric literal

2 In the following listing, place an NL before those expressions that can serve as numeric literals in a COBOL program, an FC for figurative constants, and a NON-L for nonnumeric literals; leave a blank for expressions not exemplifying any of the classes of constants.

a __________ 'DEPRECIATION SCHEDULE'
b __________ '12%'
c __________ 237
d __________ INTEREST-DUE
e __________ 125.
f __________ ZEROS
g __________ 25.32

h ________ SPACES
i ________ 100.0
j ________ 'SPACE'
k ________ '325'

a NON-L
b NON-L
c NL
d *no quotation marks*
e *cannot end with a decimal point*
f FC
g NL
h FC
i NL
j NON-L
k NON-L

SAMPLE COBOL PROGRAM

The function of the sample program is to read source records from a customer file and to print a report as illustrated in Figure 1-5. Each source record consists of 80 characters (bytes) and contains the following data fields:

Columns 1-13: Customer name.
Columns 14-16: Number of units sold for a particular product.
Columns 17-80: Other data not used by this program.

Figure 1-6 presents a flowchart for the programming project. Study this flowchart to gain an understanding of the sequence of tasks included in the project.

Figure 1-7 presents the complete COBOL program. As is true for all COBOL programs, it consists of four divisions. The IDENTIFICATION DIVISION is written first, and it consists of one paragraph-name, in this case PROGRAM-ID, which identifies this program by the name CUSTRPRT.

The ENVIRONMENT DIVISION is written next. The basic function of this division is to specify the hardware required for this program.

The DATA DIVISION follows. It is used to identify the data-names that will be used in the program, as well as their data characteristics and hierarchical structure.

The last division is the PROCEDURE DIVISION, which constitutes the executable part of the program. In this division the programmer writes the specific instructions to be carried out by the computer. Note that the instructions have the appearance of everyday language, which illustrates the self-documentation feature of the COBOL language.

```
CUSTOMER NAME     UNITS SOLD       } Report Heading

ADAMS                   100        )
BROWN                    75        )
GROVER                   30        )
MOORE                    25        } Customer Detail
PETERSON                 60        )
WILLIAMS                 10        )

TOTAL                   300        } Report Footing
```

FIGURE 1-5
DESIRED OUTPUT FOR THE REPORT PROGRAM.

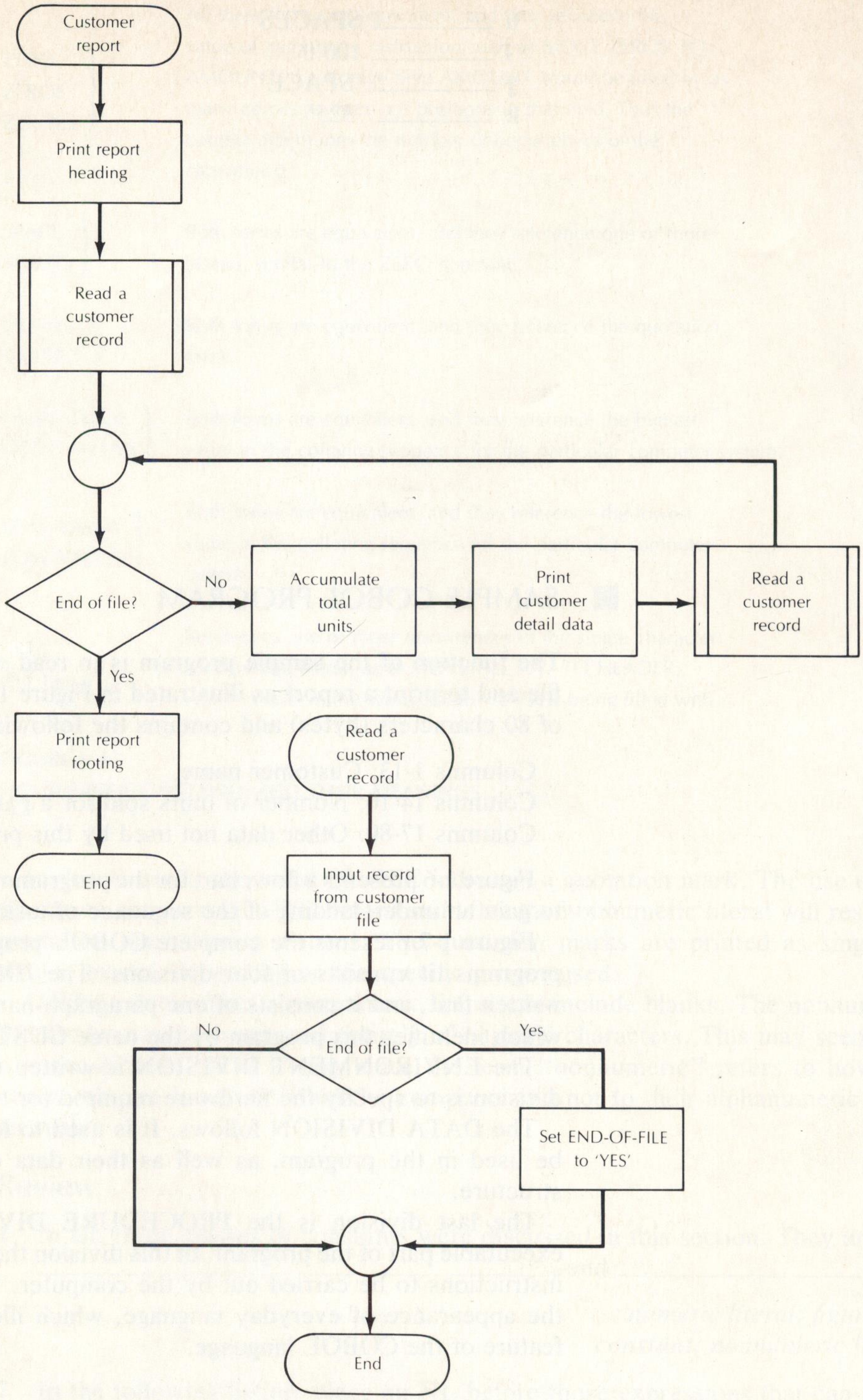


FIGURE 1-6
FLOWCHART FOR THE REPORT PROGRAM.

Comments along the right side of the sample program in Figure 1-7 explain the meaning and purpose of each group of statements. Notice that the DATA DIVISION is rather detailed. It has been designed to be a complete and clear description of all the data fields used in the program. When one needs to refer to a program written by that person months or years earlier, or to a program written by another individual, a complete data description is of tremendous help in avoiding ambiguities and confusion. This feature would be more obvious in

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CUSTRPRT.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-48Ø.                         This section identifies the
 OBJECT-COMPUTER. ABC-48Ø.                         computer system being used.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
 SELECT CUSTOMER-FILE ASSIGN TO  device.           Designation of the two files
 SELECT REPORT-FILE   ASSIGN TO  printer.          and their hardware assignment.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  CUSTOMER-FILE                                 Description of the input
     LABEL RECORDS ARE OMITTED                     customer data file.
     RECORD CONTAINS 8Ø CHARACTERS
     DATA RECORD IS CUSTOMER-RECORD.
 Ø1  CUSTOMER-RECORD.
     Ø2 CUSTOMER-NAME         PIC X(13).
     Ø2 CUSTOMER-UNITS        PIC 999.
     Ø2 FILLER                PIC X(64).
*
 FD  REPORT-FILE                                   Description of the output
     LABEL RECORDS ARE OMITTED                     report file.
     RECORD CONTAINS 132 CHARACTERS
     DATA RECORD IS REPORT-RECORD.
 Ø1  REPORT-RECORD            PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 Ø1  END-OF-FILE              PIC XXX VALUE 'NO '.     A "flag" field used to
*                                                      signal the end of file.
 Ø1  REPORT-HEADING           PIC X(31)
      VALUE 'CUSTOMER NAME        UNITS SOLD'.         A record containing the
*                                                      report heading.
Ø1  REPORT-FOOTING.
    Ø2 FILLER             PIC X(5)  VALUE 'TOTAL'.     A record containing the
    Ø2 FILLER             PIC X(18) VALUE SPACES.      report footing.
    Ø2 REPORT-TOTAL-UNITS      PIC ZZZ9.
*
 Ø1  REPORT-DETAIL-RECORD.                             A record consisting of
     Ø2 REPORT-CUST-NAME     PIC X(13).                the report detail data
     Ø2 FILLER               PIC X(1Ø)  VALUE SPACES.  fields.
     Ø2 REPORT-UNITS         PIC ZZZ9.
*
 Ø1  TOTAL-UNITS             PIC 9999 VALUE ZERO.      The units accumulator.
 PROCEDURE DIVISION.
*
 PROGRAM-SUMMARY.
*                                                      Declare the input/output
     OPEN INPUT   CUSTOMER-FILE                        files and get them open.
          OUTPUT  REPORT-FILE.
*
     MOVE REPORT-HEADING TO REPORT-RECORD              Print the report heading
     WRITE REPORT-RECORD AFTER ADVANCING PAGE.         on a new page.
*
     MOVE SPACES TO REPORT-RECORD                      Print a blank line to
     WRITE REPORT-RECORD AFTER ADVANCING 1 LINE        double-space.
*
     PERFORM READ-CUSTOMER-RECORD.                     Read the (first) customer
*                                                      record.
*
     PERFORM PROCESS-CUSTOMER-DETAIL                   Keep processing customers
          UNTIL END-OF-FILE = 'YES'.                   till the end of the file.
```

FIGURE 1-7
SAMPLE COBOL PROGRAM.

```
       *
           MOVE TOTAL-UNITS TO REPORT-TOTAL-UNITS        Print the report footing
           MOVE REPORT-FOOTING TO REPORT-RECORD          double spacing after the
           WRITE REPORT-RECORD AFTER ADVANCING 2 LINES.  last detail line.
       *
           CLOSE CUSTOMER-FILE                           Close the files.
                 REPORT-FILE.
       *
           STOP RUN.                                     Terminate the program.
       *
       READ-CUSTOMER-RECORD.
       *                                                 Read a record from the
           READ CUSTOMER-FILE RECORD                     customer file.  If it is
                AT END MOVE 'YES' TO END-OF-FILE.        the end set a flag to YES.
       *
       PROCESS-CUSTOMER-DETAIL.
       *
           ADD CUSTOMER-UNITS TO TOTAL-UNITS             Accumulate total units.
       *
           MOVE CUSTOMER-NAME TO REPORT-CUST-NAME        Move data from the input
           MOVE CUSTOMER-UNITS TO REPORT-UNITS           record to report detail.
       *
           MOVE REPORT-DETAIL-RECORD TO REPORT-RECORD    Print a customer detail
           WRITE REPORT-RECORD AFTER ADVANCING 1 LINE.   line.
       *
           PERFORM READ-CUSTOMER-RECORD.                 Read (another) customer
                                                         record.
```

FIGURE 1-7 *(Continued)*
SAMPLE COBOL PROGRAM.

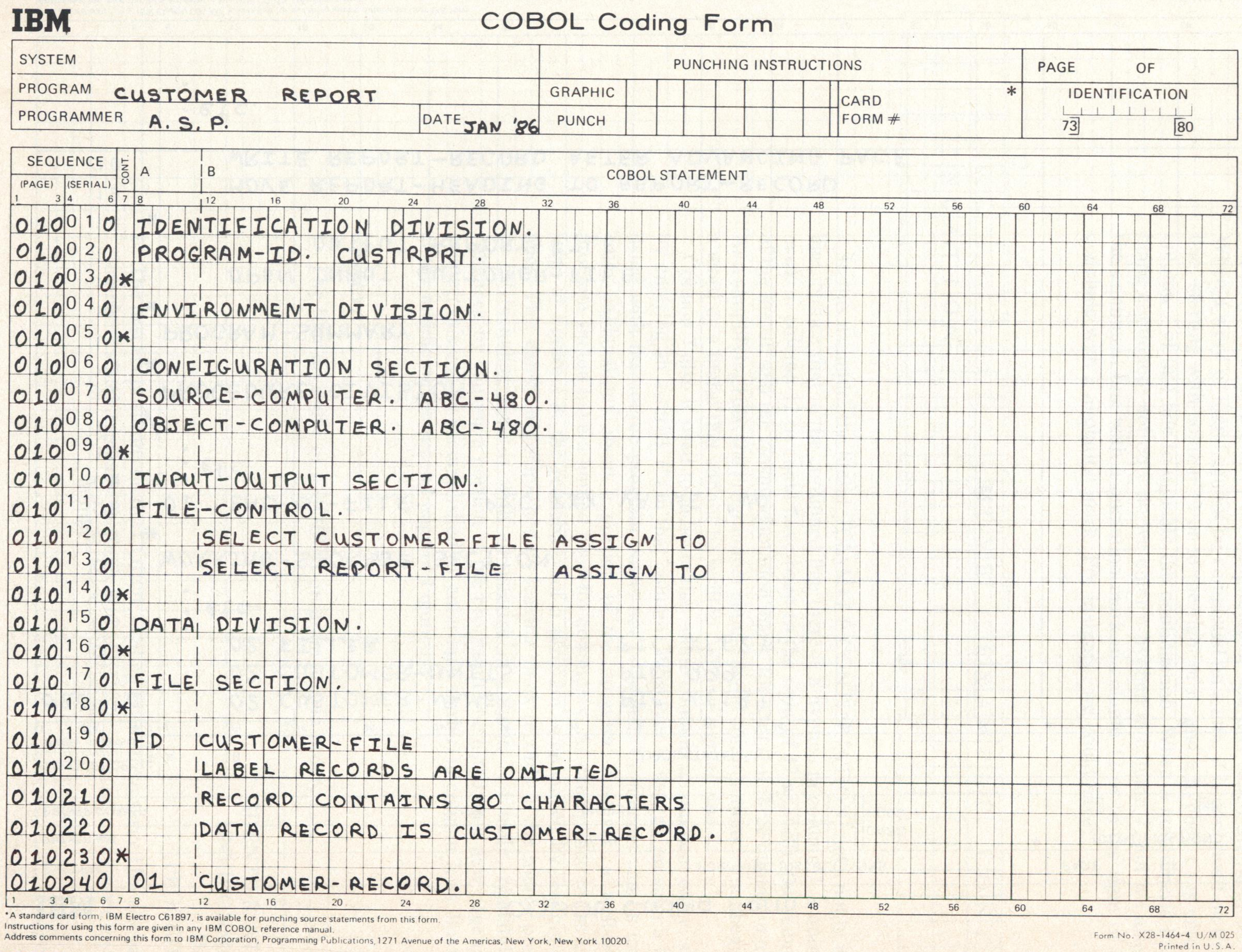
our example if the project were more like typical business programs, which generally consist of several hundred or even several thousand program lines. Of course, our purpose in a first course is to learn the language with respect to relatively simple projects, rather than implementing complex programming objectives. Nevertheless, COBOL was designed to be particularly useful for complex projects. It is a language for *managed* program development that may include the work of teams of programmers, thereby making possible the timely completion of projects.

Returning to Figure 1-7, notice that the PROCEDURE DIVISION is rather straightforward and corresponds closely to the flowchart in Figure 1-6. Again, comments are included along the right side of the figure to explain the purpose of each program statement. Even though you are not yet familiar with COBOL commands and statement formats at this point, read each PROCEDURE DIVISION statement to gain a general understanding of this program.

THE COBOL CODING FORM

A special coding form used in writing COBOL source programs is illustrated in Figure 1-8, on which the program described in the preceding section is entered. This form has been prepared to coincide with the standard 80-column format of the punched card. Of course, nowadays visual display keyboard terminals are most often used to input COBOL programs into a computer.

The first six positions of the COBOL Coding Form are reserved for the optional sequence number. The programmer may assign a sequence number to each program line so that they are numbered in order. A common practice has been to use the first three columns as a page number corresponding to the number of coding form pages used. Then the next three columns indicate line numbers, such as 010, 020, 030, and so on. Gaps are left in the sequence so that when program changes are made, new lines can be inserted without disrupting the previous sequence. For instance, a new line could be inserted between the second and third lines by assigning a line number of 025.



IBM COBOL Coding Form

SYSTEM

PROGRAM CUSTOMER REPORT

PROGRAMMER A. S. P. DATE JAN '86

PUNCHING INSTRUCTIONS: GRAPHIC / PUNCH / CARD FORM # *

PAGE OF

IDENTIFICATION 73 80

SEQUENCE (PAGE) (SERIAL) | CONT. | A | B | COBOL STATEMENT

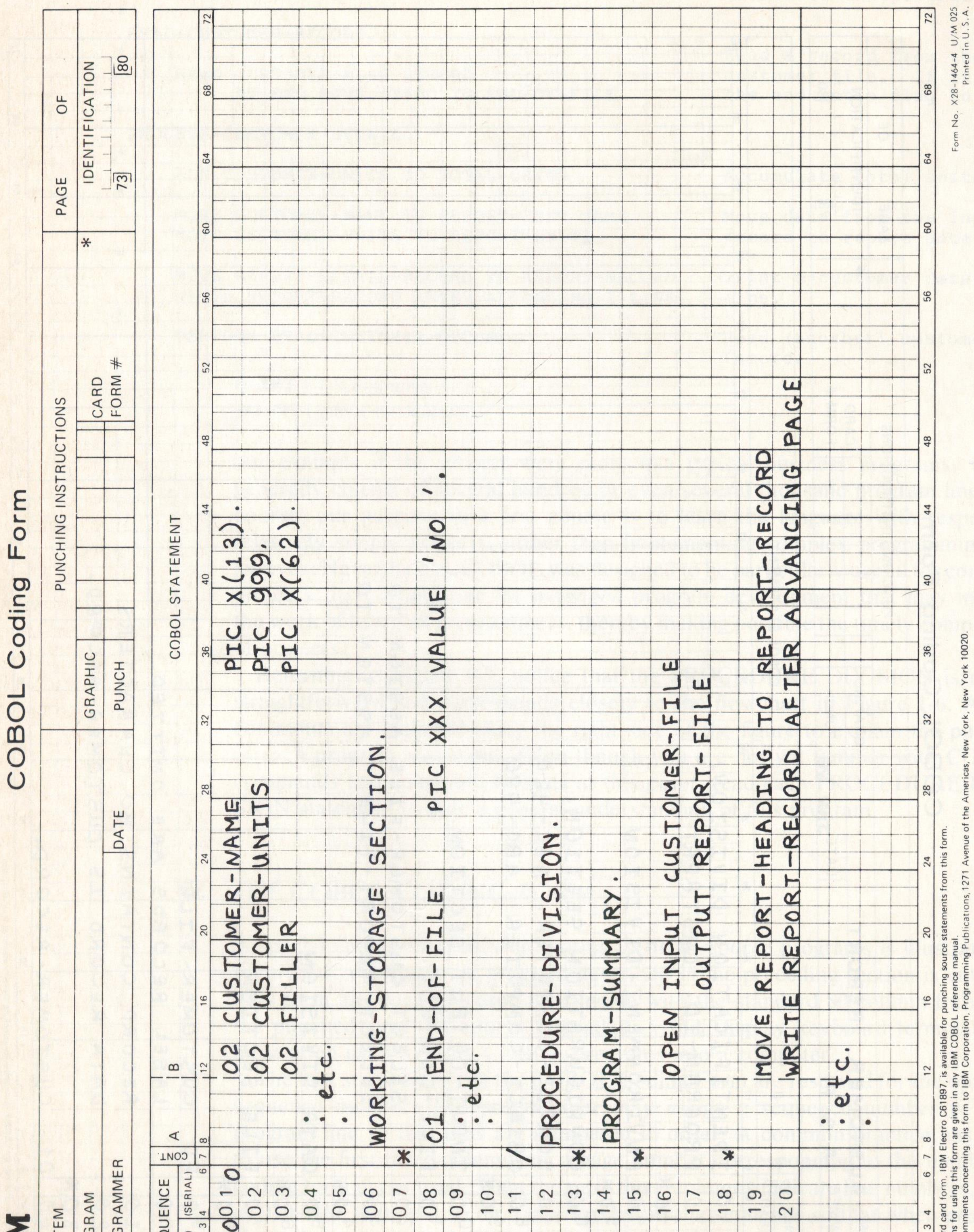
```
010010 IDENTIFICATION DIVISION.
010020 PROGRAM-ID. CUSTRPRT.
010030*
010040 ENVIRONMENT DIVISION.
010050*
010060 CONFIGURATION SECTION.
010070 SOURCE-COMPUTER. ABC-480.
010080 OBJECT-COMPUTER. ABC-480.
010090*
010100 INPUT-OUTPUT SECTION.
010110 FILE-CONTROL.
010120     SELECT CUSTOMER-FILE ASSIGN TO
010130     SELECT REPORT-FILE   ASSIGN TO
010140*
010150 DATA DIVISION.
010160*
010170 FILE SECTION.
010180*
010190 FD  CUSTOMER-FILE
010200     LABEL RECORDS ARE OMITTED
010210     RECORD CONTAINS 80 CHARACTERS
010220     DATA RECORD IS CUSTOMER-RECORD.
010230*
010240 01  CUSTOMER-RECORD.
```

*A standard card form, IBM Electro C61897, is available for punching source statements from this form.
Instructions for using this form are given in any IBM COBOL reference manual.
Address comments concerning this form to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

Form No. X28-1464-4 U/M 025
Printed in U.S.A.

FIGURE 1-8

COBOL CODING FORM WITH SAMPLE COBOL PROGRAM.



IBM COBOL Coding Form

SYSTEM | PROGRAM | PROGRAMMER | DATE | PUNCHING INSTRUCTIONS: GRAPHIC, PUNCH | CARD FORM # | * | PAGE OF | IDENTIFICATION 73 80

SEQUENCE (PAGE) (SERIAL) | CONT. | A | B | COBOL STATEMENT

```
020010  02 CUSTOMER-NAME      PIC X(13).
   02   02 CUSTOMER-UNITS     PIC 999.
   03   02 FILLER             PIC X(62).
   04    :
   05    : etc.
   06 WORKING-STORAGE SECTION.
   07*
   08 01 END-OF-FILE    PIC XXX VALUE 'NO '.
   09    : etc.
   10
   11/
   12 PROCEDURE-DIVISION.
   13*
   14 PROGRAM-SUMMARY.
   15*
   16     OPEN INPUT CUSTOMER-FILE
   17          OUTPUT REPORT-FILE
   18*
   19     MOVE REPORT-HEADING TO REPORT-RECORD
   20     WRITE REPORT-RECORD AFTER ADVANCING PAGE
          : etc.
```

*A standard card form, IBM Electro C61897, is available for punching source statements from this form. Instructions for using this form are given in any IBM COBOL reference manual. Address comments concerning this form to IBM Corporation, Programming Publications, 1271 Avenue of the Americas, New York, New York 10020.

Form No. X28-1464-4 U/M 025 Printed in U.S.A.

FIGURE 1-8 *(Continued)*
COBOL CODING FORM WITH SAMPLE COBOL PROGRAM.

The practice of using sequence numbers in a COBOL program is often unnecessary. Most programs are now being keyed in using a text editor or word processing program. Such programs allow for each insertion and deletion of lines and automatically keep track of line numbers. Therefore, it usually is not necessary to be involved in the cumbersome practice of assigning six-digit line sequence numbers. The procedure was required when punched cards were in use and it was important to identify each physical card by an appropriate sequence number.

Column 7 is used mainly for continuation, and it is called the *indicator area*. When a word or literal cannot be completed on a line, it is continued on the next line, starting with column 12 or to the right of it, and a hyphen (-) is entered in column 7 of that line to indicate the continuation. When a nonnumeric literal is being continued, not only do we enter a hyphen in column 7, but we also start the continued line with a quotation mark in column 12 or to the right of it and conclude with a quotation mark. For instance, we could have:

```
7    12                                           72
                  MOVE 'EXAMPLE CONTI
     'NUATION'
```

In this example, the literal 'EXAMPLE CONTINUATION' does not fit on the line. Notice that in the continued line there is a hyphen in column 7, a quotation mark in column 12 (it could have been to the right of 12 as well), the remainder of the literal, and the closing quotation mark, for a total of three quotation marks. As a simplifying rule, the programmer should avoid continuations by not beginning a word or a literal that cannot be written fully on a given line.

Column 7 also is used to indicate that a line contains a comment entry, shown by entering an asterisk (*) in that column. Whatever is written on such a line is listed with the source program but is not compiled. Comments can be used to enter explanations about a portion of the program; however, a well-written program should have a limited need for comments. As exemplified in the coding form in Figure 1-8, the readability of a program is enhanced by leaving blank lines. This can be accomplished by leaving blank records in the source program or by including records with an asterisk in column 7. Finally, readability is enhanced by causing a portion of the program to be listed on a new page on the printer; this can be accomplished by entering a slash(/) in column 7, as illustrated at the middle of the second page of Figure 1-8.

Review

1 The four divisions of a COBOL program are, in order, ________________, ________________, ________________, and ________________.

IDENTIFICATION; ENVIRONMENT; DATA; PROCEDURE

2 The first six positions on the COBOL Coding Form are reserved for a sequence number, which typically includes a(n) ________________ number and a(n) ________________ number.

page; line

3 When a word or literal cannot be completed on a given line, a(n) ________ is entered in column 7 of [that line / the line on which the word is continued].

hyphen; the line on which the word is continued

4 When a line contains only a comment entry, a(n) ________________ is entered in column 7 of that line.

asterisk

5 When a line is to be printed on a new page in the printer listing of the program, a(n) ________________ is entered in column 7 of that line.

slash

AN OUTLINE FOR YOUR COBOL PROGRAMS

As a student learning the COBOL language, you will probably find it convenient to create a program outline similar to the one presented in Figure 1-9. Using a text editor, you can create such an outline and save it as a text file. Then whenever you are about to key in a new program, copy the outline and add the appropriate instructions. In Figure 1-9, blank spaces indicate the locations in the program of required data, names, etc.

Using a program outline will save you program entry time as well as help to avoid the inadvertent omission of required keywords or sections.

THE OPERATING SYSTEM AND THE EXECUTION OF COBOL PROGRAMS

All modern computer systems are controlled by an *operating system* that consists of a set of programs designed to facilitate the automatic operation of the computer system and to reduce the programming task for system users. The heart of an operating system is the resident *supervisor* (*executive* and *monitor* are synonymous terms), which is a program that acts as the control module in the operating system. The computer operator loads the supervisor as the first task "each morning," and from then on the supervisor controls the operation, with human intervention being required only to implement some exceptional procedure. The supervisor loads and executes other programs that either are part of the operating system or are submitted by users. When a program has been executed completely, control is resumed by the supervisor, which executes control functions until it transfers control to another program. When the machine is idle, the supervisor is in a "wait" state, waiting for the next program to be executed.

The motivation for the development of operating systems is related to the basic difference in speed between computer systems and human operators. In the early days of computing, a human operator took each program to be run, loaded it into the machine, and removed the associated output prior to loading the next program. As computers became faster and the number of users multiplied, it became obvious that the operation of computer systems had to be automated. In a modern computer system, this automation is achieved by the operating system. Human operators continuously input the programs to be processed (often

```
IDENTIFICATION DIVISION.
PROGRAM-ID. __________.

ENVIRONMENT DIVISION.

CONFIGURATION SECTION.

SOURCE-COMPUTER. ____________.
OBJECT-COMPUTER. ____________.

INPUT-OUTOUT SECTION.
FILE-CONTROL.
    SELECT ______________ ASSIGN TO ____________
    SELECT ______________ ASSIGN TO ____________.

DATA DIVISION.

FILE SECTION.

FD  ______________

    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS ___ CHARACTERS
    DATA RECORD IS ______________.

01  ______________

FD  ______________

    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS ___ CHARACTERS
    DATA RECORD IS ______________.

01  ______________

WORKING-STORAGE SECTION.

PROCEDURE DIVISION.
```

FIGURE 1-9
AN OUTLINE FOR COBOL PROGRAMS.

simultaneously in different locations), while the operating system maintains control over the program-to-program execution sequence, checks availability of resources (tapes, disk space, printers) and, when needed, transfers control from one program to another in an attempt to maximize *throughput*.

The development of operating systems represented a major conceptual breakthrough and has required extensive programming efforts. In fact, programming is differentiated into systems programming (systems software is a synonym) and applications programming. *Systems programs* relate to operating systems and include supervisors, language processors, data management such as disk-space allocation, and utilities. (*Utilities* are programs that perform such standard tasks as copying data from disk to tape and dumping a tape on the printer.) *Applications programs*, on the other hand, are oriented toward the use of the computer for some data processing task. A COBOL programmer is an applications programmer. All applications programs require the use of systems programs in their processing, and therefore it is essential to have a basic understanding of the operating system of the computer with which the applications programmer is working.

Operating systems consist of hundreds of thousands, or even millions, of program instructions and require millions of dollars and many years of effort to develop. For these reasons, operating systems are considered an essential part

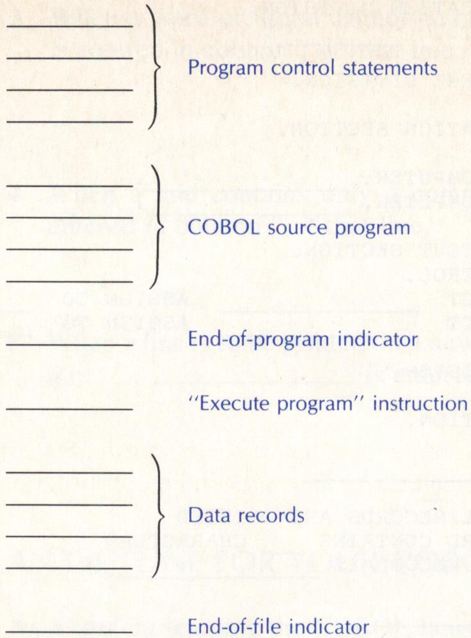


FIGURE 1-10
TYPICAL COBOL PROGRAM SETUP.

of the equipment, are developed and maintained by each computer manufacturer, and are provided as part of the total cost of the equipment. Under such circumstances, you can see why each manufacturer has an operating system that differs from the others and often has different operating systems for different computer models. Because of these differences, we cannot explain specifics in this text. We will, however, discuss the general concepts that relate COBOL programming to operating systems, and we will assume that the reader has access to information about the specific operating system used by his or her computer installation.

In order for a COBOL program to be run, the programmer needs to communicate certain things to the operating system. Typical areas of information include the following:

Identify the user as a legitimate user of the machine.
Indicate the fact that a COBOL program is being used. (In a typical installation, programs are written in several languages and they need to be differentiated.)
Request compilation.
Indicate whether the compilation output (object program) is to be saved on disk, tape, etc.
Request the use of tapes, disks, printers, and other devices.
Request execution of the compiled program.

The communication process between the programmer and the operating system is effected by means of system commands known collectively as the Job Control Language (JCL). The JCL, of course, is not standardized. Figure 1-10 portrays a typical COBOL program setup. In the past, the JCL statements were punched onto cards referred to as *program control cards*. Although punched cards as such are seldom used anymore, the practice still is to refer to "cards." In general, the first card identifies the user by account number or by name and may set parameters on maximum time, maximum number of pages, and the like. Then there are one or more JCL statements to invoke the compiler and to indicate the disposition of the object program. The source program follows, possibly with an end-of-

program indicator. Then there are one or more JCL statements to ready the object program for execution. The "execute" type of instruction often is followed by the data and by an end-of-file indicator.

Overall, then, program control statements serve to transmit certain categories of information to the operating system. On the basis of this information, execution of the program is scheduled and monitored by the supervisor, which is the systems program concerned with controlling the operation of the computer.

Review

1 The entire set of programs designed by the computer manufacturer to facilitate the automatic operation of the computer is called the ________________.

operating system

2 The particular program in the operating system that serves as a control module in the selection and scheduling of programs for execution is called the ________________.

supervisor (or executive or monitor)

3 The types of computer programs concerned with such functions as disk-space allocation and copying data from cards to tape are called ________________ programs.

systems (or utility)

4 The types of computer programs concerned with actual data processing tasks are called ________________ programs.

applications

5 A programmer who uses COBOL as the programming language is a(n) [systems / applications] programmer.

applications

6 The communication process between the programmer and the operating system is carried out through the use of a set of commands called the ________________.

Job Control Language (JCL)

7 The first set of statements in a typical COBOL program setup are the ________________ statements.

program control

8 In the typical COBOL program setup, the special record that follows the data records, and is the last item in the program setup, is the ________________ indicator.

end-of-file

EXERCISES

1.1 Explain what is meant by a computer language.

1.2 Name and briefly discuss three levels of computer languages.

1.3 Explain these three related concepts: source program, object program, and compiler. How do these three interrelate in a typical COBOL program run?

1.4 What does the following statement mean: "COBOL programs should be substantially self-documenting"?

1.5 What is the difference between data-names and constants in COBOL?

1.6 Explain what is meant by a figurative constant and name some such constants.

1.7 Give the name and the role of each of the divisions that make up a COBOL program.

1.8 What is the operating system of a computer? How do program control cards relate to such a system?

1.9 Programming is not a single step but is best thought of as a process. Describe the steps included in the programming process.

1.10 Modify the sample program in Figure 1-7 so that the report heading consists of the following three lines, including the line made up of dashes:

```
                    CUSTOMER REPORT
CUSTOMER NAME                                   UNITS SOLD
--------------------------------------------------------------
```

1.11 Enter and run the sample program given in Figure 1-7, using data of your own choice. Een though this exercise does not require programming effort on your part, it will provide you with the opportunity to familiarize yourself with the program-submission procedures of your computer system.

2 AN ELEMENTARY SET OF COBOL INSTRUCTIONS

INTRODUCTION

This chapter presents a limited set of COBOL instructions so that you can write simple but complete programs. Additional language features will continue to be introduced in subsequent chapters.

We present the elementary set of instructions by means of a sample program, a somewhat expanded version of the sample program in Chapter 1. As before, the data is a file of customer records, with each record containing a customer name and number of units sold of a particular product. The sample program in the present chapter is designed to read each record in the file and to produce the sample report portrayed in Figure 2-1.

The NET SALES column in Figure 2-1 results from the following computational basis: If the number of units sold is 20 or less, the unit price is \$10.00. If the number of units sold is greater than 20, the unit price is discounted to \$9.50. The net sales value then is computed by multiplying the number of units sold by the appropriate unit price.

Figure 2-2 presents a listing of the sample program. This program will be referenced throughout the chapter to allow the reader to relate the various types of language statements to a concrete case.

IDENTIFICATION DIVISION

The function of the IDENTIFICATION DIVISION is to supply information about the program to others who may read or use the program. On the COBOL Coding Form, we start in column 8 with the words INDENTIFICATION DIVISION. The first and only required paragraph is the PROGRAM-ID, which

CUSTOMER NAME	UNITS SOLD	NET SALES
ADAMS	100	950.00
BROWN	75	712.50
GROVER	30	285.00
MOORE	25	237.50
PETERSON	60	570.00
WILLIAMS	10	100.00
TOTAL	300	2855.00

FIGURE 2-1

SAMPLE SALES REPORT.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  CUSTRPRT.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CUSTOMER-FILE ASSIGN TO File-name/Device.
     SELECT REPORT-FILE   ASSIGN TO Printer.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  CUSTOMER-FILE
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS CUSTOMER-RECORD.
 01  CUSTOMER-RECORD.
     02 CUSTOMER-NAME          PIC X(13).
     02 CUSTOMER-UNITS         PIC 999.
     02 FILLER                 PIC X(64).
*
 FD  REPORT-FILE
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 132 CHARACTERS
     DATA RECORD IS REPORT-RECORD.
 01  REPORT-RECORD             PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE               PIC XXX VALUE 'NO'.
*
 01  REPORT-HEADING            PIC X(40)
       VALUE 'CUSTOMER NAME     UNITS SOLD    NET SALES'.
*
 01  REPORT-FOOTING.
     02 FILLER                    PIC X(5)  VALUE 'TOTAL'.
     02 FILLER                    PIC X(14) VALUE SPACES.
     02 REPORT-TOTAL-UNITS        PIC ZZZ9.
     02 FILLER                    PIC X(9)  VALUE SPACES.
     02 REPORT-TOTAL-NET-SALES PIC ZZZ9.99.
*
 01  REPORT-DETAIL-RECORD.
     02 REPORT-CUST-NAME       PIC X(13).
     02 FILLER                 PIC X(7)  VALUE SPACES.
     02 REPORT-UNITS           PIC ZZ9.
     02 FILLER                 PIC X(9)  VALUE SPACES.
     02 REPORT-NET-SALES       PIC ZZZ9.99.

 01  NET-SALES                 PIC 9999V99.
```

FIGURE 2-2

SAMPLE COBOL PROGRAM.

```
      *
       01  REPORT-TOTALS.
           02 TOTAL-UNITS             PIC 9999    VALUE ZERO.
           02 TOTAL-NET-SALES         PIC 9999V99 VALUE ZERO.
      *
       01  PRICES.
           02 PRICE                   PIC 99V99.
           02 FULL-PRICE              PIC 99V99   VALUE 10.00.
           02 DISCOUNT-PRICE          PIC 99V99   VALUE 9.50.
      /
       PROCEDURE DIVISION.
      *
       PROGRAM-SUMMARY.
      *
           OPEN INPUT CUSTOMER-FILE
                OUTPUT REPORT-FILE.
      *
           MOVE REPORT-HEADING TO REPORT-RECORD
           WRITE REPORT-RECORD AFTER ADVANCING PAGE.
           MOVE SPACES TO REPORT-RECORD
           WRITE REPORT-RECORD AFTER ADVANCING 1 LINE
      *
           PERFORM READ-CUSTOMER-RECORD.
      *
           PERFORM PROCESS-CUSTOMER-DETAIL
                   UNTIL END-OF-FILE = 'YES'.
      *
           MOVE TOTAL-UNITS TO REPORT-TOTAL-UNITS
           MOVE TOTAL-NET-SALES TO REPORT-TOTAL-NET-SALES
           MOVE REPORT-FOOTING TO REPORT-RECORD
           WRITE REPORT-RECORD AFTER ADVANCING 2 LINES.
      *
           CLOSE CUSTOMER-FILE
                 REPORT-FILE.
      *
           STOP RUN.
      *
       READ-CUSTOMER-RECORD.
      *
           READ CUSTOMER-FILE RECORD
               AT END MOVE 'YES' TO END-OF-FILE.
      *
       PROCESS-CUSTOMER-DETAIL.
      *
           ADD CUSTOMER-UNITS TO TOTAL-UNITS
      *
           IF CUSTOMER-UNITS > 20
               MOVE DISCOUNT-PRICE TO PRICE
           ELSE
               MOVE FULL-PRICE TO PRICE.
      *
           MULTIPLY CUSTOMER-UNITS BY PRICE GIVING NET-SALES.
      *
           ADD NET-SALES TO TOTAL-NET-SALES
      *
           MOVE CUSTOMER-NAME TO REPORT-CUST-NAME
           MOVE CUSTOMER-UNITS TO REPORT-UNITS
           MOVE NET-SALES TO REPORT-NET-SALES
      *
           MOVE REPORT-DETAIL-RECORD TO REPORT-RECORD
           WRITE REPORT-RECORD AFTER ADVANCING 1 LINE.
      *
           PERFORM READ-CUSTOMER-RECORD.
```

FIGURE 2-2 *(Continued)*
SAMPLE COBOL PROGRAM.

is followed by the program-name chosen by the programmer. As in the example, we may have:

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CUSTRPRT.
```

In this case, the word CUSTRPRT is the name the programmer has chosen to identify the program. This name must start with an alphabetic character and may consist of up to 30 alphabetic or numeric characters unless the specific compiler limits the number to fewer characters. The two lines shown in the example are sufficient content for the IDENTIFICATION DIVISION. All other paragraphs are optional but, if they are used, must be written in the order shown. The following example includes optional paragraphs. The underlined words are COBOL reserved words, which are required; the other words are a matter of the programmer's choice.

<u>IDENTIFICATION</u> <u>DIVISION</u>.
<u>PROGRAM-ID</u>. SALARY.
<u>AUTHOR</u>. LEE WALTERS.
<u>INSTALLATION</u>. XYZ CORPORATION.
<u>DATE-WRITTEN</u>. JANUARY 14, 1986.
<u>DATE-COMPILED</u>.
<u>SECURITY</u>. THIS PROGRAM RESTRICTED TO PERSONNEL WHO HAVE BEEN CLEARED
    BY THE CONTROLLER'S OFFICE.

All paragraph-names start in column 8 and, as indicated above, are all optional with the exception of PROGRAM-ID. The compiler does not process what follows the COBOL words but only prints this content. Thus, after DATE-WRITTEN we could have written DURING THE SPRING OF 86. The compiler derives no more meaning from JANUARY or from SPRING than from the nonsense word AMZKAB; therefore, the programmer should be concerned simply with choosing verbal descriptions that will be meaningful to the potential readers of the program.

Note that the DATE-COMPILED paragraph is left blank. The compiler will insert the actual date, and the source listing will include that date.

An entry in the IDENTIFICATION DIVISION may extend to more than one line, as illustrated in the case of the SECURITY paragraph. In such a case, the lines subsequent to the first line must all start in column 12 or to the right of column 12. The PROGRAM-ID paragraph, however, is restricted to one word, which must not exceed 30 characters in length.

Review

1 The order in which the four divisions of a COBOL program appear is: ________________, ________________, ________________, and ________________.

IDENTIFICATION;
ENVIRONMENT;
DATA;
PROCEDURE

2 Other than the division name itself, the only paragraph required in the IDENTIFICATION DIVISION is the one named ________________.

PROGRAM-ID

3 Although other entries in the IDENTIFICATION DIVISION may extend to more than one line, the PROGRAM-ID paragraph is normally restricted to

one word which must not exceed ______________ (number) characters in length.

30

4 Overall, the purpose of the IDENTIFICATION DIVISION of a COBOL program is to ______________.

describe the program to potential users (etc.)

ENVIRONMENT DIVISION

A COBOL program written for one computer normally can be processed on another computer, with the exception of the ENVIRONMENT DIVISION portion of the program. The reason for this exception is that the specified equipment to be used in running the program is described in this division. Since the equipment at different computer centers differs in terms of manufacturer and model, the ENVIRONMENT DIVISION of a COBOL program has to be tailored to each computer installation. This may seem inconvenient, but consider also the advantages of this modular design of COBOL. Since equipment differences affect only the ENVIRONMENT DIVISION, this division of the program can be changed without rewriting any of the other three divisions.

The following illustration of an ENVIRONMENT DIVISION is modified from the sample program in Figure 2-2:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT CUSTOMER-FILE ASSIGN TO CUSTFILE.
    SELECT REPORT-FILE ASSIGN TO PRINTER.
```

There are two sections in this example division, the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION. The SOURCE-COMPUTER and OBJECT-COMPUTER paragraphs serve documentation purposes. ABC-480 is fictitious in the example and would be replaced by a manufacturer's name and model number.

In this example, there is one paragraph in the INPUT-OUTPUT SECTION: the FILE-CONTROL paragraph. The SELECT statement identifies the name of a file, in this case CUSTOMER-FILE. This file-name is the programmer's choice and is formed in compliance with COBOL name formation rules (although some compilers place further restrictions, such as that the first 12 characters must be unique). The ASSIGN statement declares that this file will be associated with the file identification that follows, in this case CUSTFILE. Similarly, REPORT-FILE identifies another file-name, and the hardware device with which this file will be associated is PRINTER (thus, this file will include the output information). Device names are neither COBOL words nor programmer supplied; they are *implementor-names*, which means they are specific to the compiler used. Sometimes the implementor-names can be rather cryptic, as in:

```
SELECT CUSTFILE ASSIGN TO SYS005-UR-2540R-S.
```

Each computer system has its own way of designating computer files. Therefore in this book such entries usually are presented in a general conceptual form rather than in a computer-specific form.

Review

1 Of the four divisions of a COBOL program, the one written with an orientation to a particular computer system and its equipment is the ______________ DIVISION.

ENVIRONMENT

2 The two sections of the ENVIRONMENT DIVISION are the CONFIGURATION SECTION and the INPUT-OUTPUT SECTION. Of these, the one that serves to identify the equipment to be used and is the only section specifically required is the ______________ SECTION.

CONFIGURATION

3 The section of the ENVIRONMENT DIVISION concerned with the assignment of specified files to particular devices is the ______________ SECTION.

INPUT-OUTPUT

4 Overall, the division of a COBOL program concerned with supplying general information about the program is the ______________ DIVISION, whereas the division that specifies the equipment to be used is the ______________ DIVISION.

IDENTIFICATION; ENVIRONMENT

DATA DIVISION

The DATA DIVISION identifies storage fields and their names. Commonly, it consists of two sections: the FILE SECTION and the WORKING-STORAGE SECTION. In its simplest form, the function of the FILE SECTION of the DATA DIVISION is to describe each file used in the program by specifying:

1 The name of the file
2 The name assigned to the record in the file
3 The data fields in the record
4 The field size and type of data in each storage field of the record.

The FILE SECTION in Figure 2-2 is:

```
FILE SECTION.
*
FD  CUSTOMER-FILE
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 80 CHARACTERS
    DATA RECORD IS CUSTOMER-RECORD.
01  CUSTOMER-RECORD.
    02  CUSTOMER-NAME       PIC  X(13).
    02  CUSTOMER-UNITS      PIC  999.
    02  FILLER              PIC  X(64).
*
FD  REPORT-FILE
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS 132 CHARACTERS
    DATA RECORD IS REPORT-RECORD.
01  REPORT-RECORD           PIC  X(132).
```

The designation FD is needed at the beginning of each file description and stands for File Description. It is always followed by a file-name, in this case CUSTOMER-FILE for the first file.

The clause LABEL RECORDS ARE OMITTED will be discussed in Chapter 9, on sequential file processing. For the time being we will use it as shown and will understand that it states that the file has no descriptive or identifying labels.

The RECORD CONTAINS . . . CHARACTERS is used to declare the size of the record. In the CUSTOMER-FILE each record is 80 characters in length, and we refer to such a record by means of the name CUSTOMER-RECORD, which is designated in the clause DATA RECORD IS CUSTOMER-RECORD.

Following the FD entry we see the record description. The 01 *level number* introduces the record-name CUSTOMER-RECORD, referring to the entire record. Then there are three entries, each starting with the level number 02. The 02 means that CUSTOMER-NAME, for example, is a subdivision of the 01 record, CUSTOMER-RECORD.

Figure 2-3 should help explain some fundamental concepts about files, records, and fields in a COBOL program. Within the program we use a file-name such as CUSTOMER-FILE. However, that file name may be designated by other means in terms of the operating system of the computer. In the example, the customer file is a disk file and is called CUSTFILE, as illustrated in Figure 2-3.

The 01 record entry describes the central storage area referenced by the record-name CUSTOMER-RECORD. Similarly, the fields CUSTOMER-NAME, CUSTOMER-UNITS, and FILLER also refer to central storage locations, as illustrated in Figure 2-3.

Assuming CUSTOMER-FILE to be an input file, we can give an instruction to copy a data record from the disk into central storage (by use of the READ verb). Subsequent to a READ operation we can then give instructions to operate

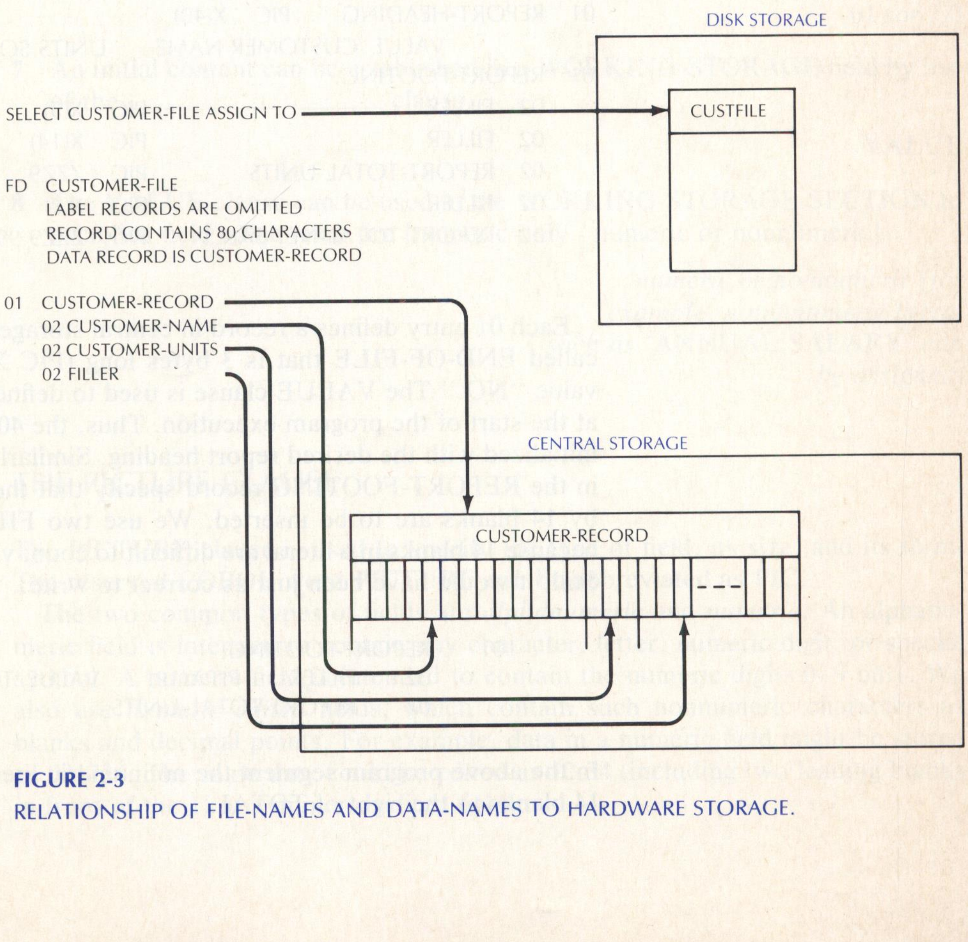


FIGURE 2-3
RELATIONSHIP OF FILE-NAMES AND DATA-NAMES TO HARDWARE STORAGE.

on the data in central storage by instructions such as ADD CUSTOMER-UNITS TO . . .

CUSTOMER-NAME and CUSTOMER-UNITS are data-names referencing fields in the record. As illustrated in Figure 2-3, CUSTOMER-NAME refers to the first 13 bytes of the CUSTOMER-REC while CUSTOMER-UNITS refers to the next 3 bytes. The size and type of each field is specified by the PIC clause, explained later in this chapter. The third field in CUSTOMER-RECORD is named FILLER.

FILLER is a COBOL reserved word. It is an optional name for a field to which no specific reference will be made in the program. The use of FILLER eliminates the need to coin more data-names than needed. Since it is not used directly, FILLER can be used to name several fields without causing any problems of unique identification. An imperative rule of the language is that all references in a program must be made to unique field names. This rule is not violated by use of the nonunique name FILLER, because this field is not referenced in the program.

The second FD entry in the sample program in Figure 2-2 describes the REPORT-FILE associated with the printer. The record-name is REPORT-RECORD, and it has no subordinate data field.

Storage fields associated with the file records receive data from, or are used to send data to, external input-output devices, such as disks, tapes, printers, and video terminals. In addition to such storage fields, there also is a need for storage of header data, and the like. The working storage section is used to define such fields.

The following excerpts from Figure 2-2 illustrate the WORKING-STORAGE SECTION:

```
WORKING-STORAGE SECTION.
01  END-OF-FILE          PIC  XXX  VALUE 'NO'.
01  REPORT-HEADING       PIC  X(40)
        VALUE 'CUSTOMER NAME      UNITS SOLD      NET SALES'.
01  REPORT-FOOTING.
    02  FILLER                    PIC  X(5)      VALUE 'TOTAL'.
    02  FILLER                    PIC  X(14)     VALUE SPACES.
    02  REPORT-TOTAL-UNITS        PIC  ZZZ9.
    02  FILLER                    PIC  X(9)      VALUE SPACES.
    02  REPORT-TOTAL-NET-SALES    PIC  ZZZ9.99.
```

Each 01 entry defines a record in central storage. The first entry defines a field called END-OF-FILE that is 3 bytes long (PIC XXX) and is initialized to the value "NO." The VALUE clause is used to define the contents of a record/field at the start of the program execution. Thus, the 40-byte REPORT-HEADING is initialized with the desired report heading. Similarly, the first two FILLER fields in the REPORT-FOOTING record specify that the characters TOTAL followed by 14 blanks are to be inserted. We use two FILLER fields for convenience, because 14 blanks in a literal are difficult to count visually and may lead to errors. Still, it would have been just as correct to write:

```
01    REPORT-FOOTING.
      02    FILLER    PIT X(19)    VALUE 'TOTAL
      02    REPORT-TOTAL-UNITS. . .
```

In the above program segment the nonnumeric literal value of FILLER contains 14 blanks to the right of TOTAL.

Review

1 Of the four divisions of a COBOL program, the one concerned with the identification and description of storage fields is the ________________ division.

DATA

2 After the DATA DIVISION and FILE SECTION have been identified in the program, the designation FD is used and is always followed by a filename. FD is a reserved COBOL word and stands for ________________ ________________.

File Description

3 After the name of the file, the next item of information given in the FD entry is the name of the ________________ contained in the file.

record

4 The level number assigned to the whole record is always ________________ (number).

01

5 All fields in the record that are directly subordinate to the overall record commonly are assigned the level number ________________ (number).

02 *or higher*

6 When a WORKING-STORAGE field is at the elementary level and therefore contains no subordinate fields, the level number assigned to the field is ____ (number).

01 (*or* 77)

7 An initial content can be established in a WORKING-STORAGE field by use of the ________________ clause.

VALUE

8 The VALUE clause can be used in the WORKING-STORAGE SECTION to establish field values that are [numeric only / numeric or nonnumeric].

numeric or nonnumeric (for example, a nonnumeric literal such as 'ANNUAL SALARY' *can be assigned)*

THE PICTURE CLAUSE

The PICTURE clause is used to specify the type of field, its size, and its form. The reserved COBOL word PICTURE can be abbreviated as PIC.

The two common types of fields are *alphanumeric* and *numeric*. An alphanumeric field is intended to contain any character, letter, numeric digit, or special symbol. A numeric field is intended to contain the numeric digits 0–9 only. We also use *numeric edited* fields, which contain such nonnumeric characters as blanks and decimal points. For example, data in a numeric field might be stored as 001234. The same data could be edited as 12.34 (including two leading blanks in front of the 1) in a numeric edited field.

We illustrate use of the PIC clause by reference to the following examples from the sample program in Figure 2-2.

02	CUSTOMER-NAME	PIC X(13)	An alphanumeric field of 13 bytes. The X character specifies an alphanumeric field and the number in parentheses specifies the size of the field.
02	CUSTOMER-UNITS	PIC 999	A numeric integer field of 3 bytes. The 9 characters specify numeric positions in the field.
01	END-OF-FILE	PIC XXX	An alphanumeric field of 3 bytes.
02	REPORT-TOTAL-UNITS	PIC ZZZ9	An *edited* numeric integer field of 4 bytes. The Z characters designate the suppression of leading zeros (if any) in the first 3 positions of the field.
02	REPORT-TOTAL-NET-SALES	PIC ZZZ9.99	An edited numeric field of 7 bytes. Zero suppression is specified for the first 3 positions. A decimal point is inserted in the fifth of the seven bytes.
02	TOTAL-NET-SALES	PIC 9999V99	A numeric field of 6 bytes. The V in column 5 specifies an *assumed* decimal point. When processing the data in this field, the last two digits are treated as decimal values. However there is no decimal point as such included in the data (it is as if a "mental note" is made as to the location of the decimal point).

There are many more types of PICTURE specifications and they will be studied in Chapter 4. For now the following rules should be learned.

- To specify an alphanumeric field we use an X picture character followed by parentheses enclosing the size of the field. Short fields may also be written by repeating the X character. For example PIC X(3) and PIC XXX are equivalent.
- To specify an integer numeric field we use one or more 9 characters. For example, a 3-digit numeric field can be specified as PIC 999. Also parentheses can be used to enclose the size of the field, so that PIC 9999 and PIC 9(4) are equivalent.
- To specify a numeric field that has decimal places, we use the V character to show the position of the decimal for purposes of doing arithmetic. For example, PIC 9V99 specifies a three-digit decimal field with the last two digits assumed to be decimal values. Also, parentheses can be used rather than repetition of the 9 character. For example, PIC 999V99 can also be written as 9(3)V99 or 9(3)V9(2).
- Numeric fields may be edited to improve the legibility of data in a report. Two editing characters are the Z and the decimal point.

  Z characters are used to suppress leading zeros. The clause PIC ZZ9 specifies that the field should be edited; any leading zeros are converted to blanks. For example, a value of 003 would be converted to 3 (a 3 with two leading blanks).

  Decimal points may be inserted into fields by using a decimal point in the appropriate position. Thus PIC 99.99 specifies a five-byte field containing two integer digits, a decimal point, and two decimal digits. We can also combine the two forms of editing to specify zero suppression and decimal point insertion as in this example: PIC ZZ9.99. Fields containing editing decimal points cannot be used for arithmetic, since a decimal point is not a numeric digit. For

computational work we specify the position of the decimal point with a V character. Then if we want to print a value in legible form to include a decimal point, we move the data from the computational to the edited field and then print the edited result.

The basic rules presented in this section are sufficient for the level of applications considered in this chapter. With these few rules we can handle many common programming situations.

Review

1 The character in the PICTURE clause used to indicate that the field can contain alphabetic, numeric, or special symbols is the ______________ character.

X

2 The character in the PICTURE clause which indicates that a field has numeric content only is the ______________ character.

9

3 The character in the PICTURE clause which indicates the position of an assumed decimal point is the ______________ character.

V

4 Either the V PICTURE character or the decimal (.) PICTURE character can be used to indicate the position of the decimal point. Of these, the one that requires use of a numeric edited field is the ______________ character.

decimal (.)

5 The PICTURE character that serves the editing function of replacing leading zeros in a value with blanks is the ______________ character.

Z

PROCEDURE DIVISION

The IDENTIFICATION, ENVIRONMENT, and DATA divisions in a COBOL program perform "housekeeping tasks" in that they provide background information so that the program can be executed after compilation. On the other hand, the instructions that directly result in execution of the program are given in the PROCEDURE DIVISION. Most of the instructions in the PROCEDURE DIVISION are instructions to operate on storage locations, or fields, that have been defined in the DATA DIVISION. Figure 2-4 illustrates the point. Storage fields are shown along with the COBOL verbs that process the illustrated contents of these fields. For instance, the customer name ANDREWS in the input file record is transferred (copied) into a WORKING-STORAGE field by the instruction:

```
MOVE CUSTOMER-NAME TO REPORT-CUST-NAME
```

Other corresponding statements taken from the sample program in Figure 2-2 and illustrated in Figure 2-4 are:

```
MOVE CUSTOMER-UNITS TO REPORT-UNITS
MULTIPLY CUSTOMER-UNITS BY PRICE GIVING NET-SALES
MOVE NET-SALES TO REPORT-NET-SALES
MOVE REPORT-DETAIL-RECORD TO REPORT-RECORD
WRITE REPORT-RECORD AFTER ADVANCING 1 LINE
```

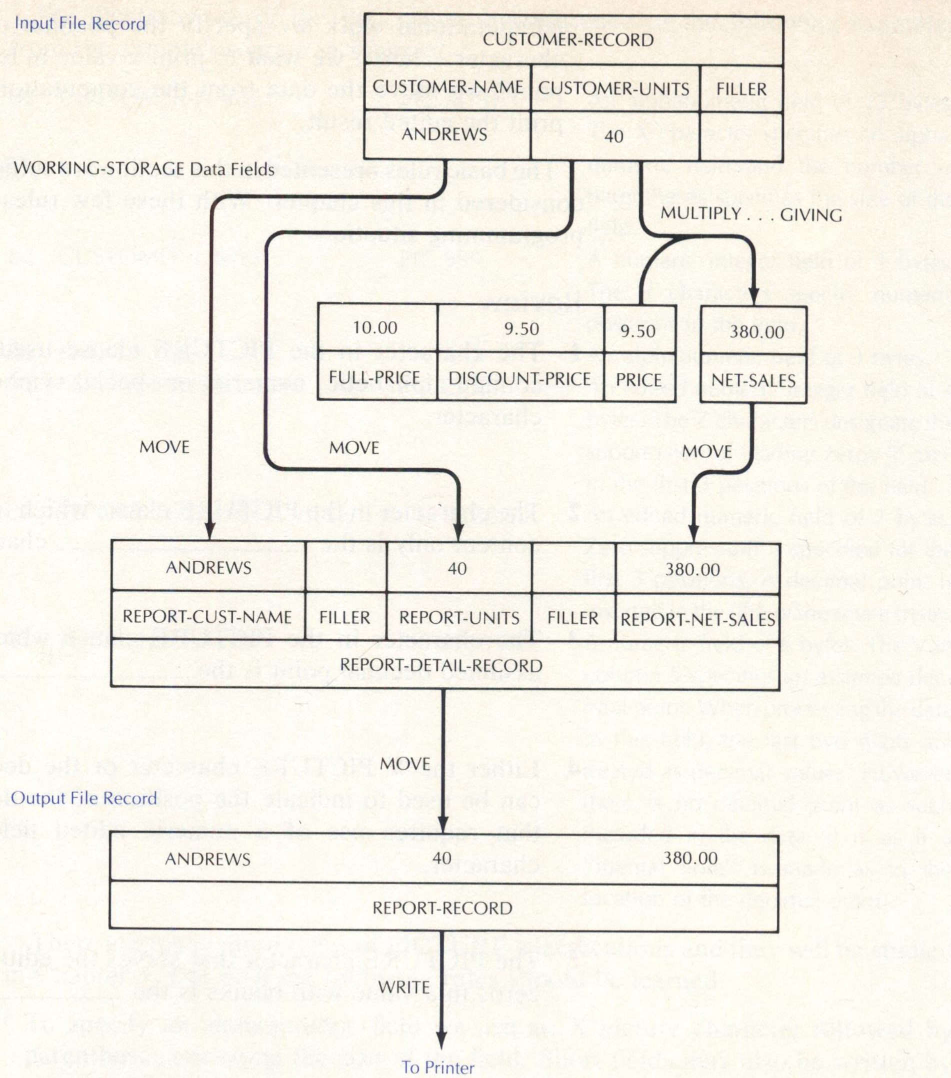


FIGURE 2-4

PARTIAL STORAGE MAP AND ILLUSTRATION FOR PROCESSING ONE CUSTOMER RECORD.

The key words PROCEDURE DIVISION identify the beginning of this division and begin at the A margin of the COBOL Coding Form, followed by a period. The division consists of paragraphs, each paragraph containing at least one sentence. Each paragraph starts with a paragraph-name beginning in column 8 (A margin). Paragraph-names are coined by the programmer following the rules of data-name formation, with one additional option: paragraph-names may be all-numeric. Sentences and statements are written in columns 12–72.

In this chapter, we present some basic instructions used in the PROCEDURE DIVISION. Additional PROCEDURE DIVISION statements are described throughout the book.

The two most commonly used types of PROCEDURE DIVISION statements are imperative and conditional statements. An imperative statement consists of a verb that indicates action, plus appropriate operands involved in the action. In this chapter, the following imperative verbs are described:

Input-output verbs: OPEN, READ, WRITE, CLOSE
Data transfer verb: MOVE
Arithmetic verbs: ADD, SUBTRACT, MULTIPLY, DIVIDE
Control verbs: PERFORM, STOP

In addition to the four types of verbs, two basic conditional expressions are covered in this chapter: The IF and the AT END conditional expressions.

Review

1 The division of a COBOL program that includes the instructions directly concerned with implementing the programming objectives of the program is the ______ DIVISION.

PROCEDURE

2 The instructions, or commands, included in the PROCEDURE DIVISION typically operate on storage fields that have previously been defined in the ______ DIVISION of the program.

DATA

3 Of the two commonly used types of statements used in the PROCEDURE DIVISION, the type that includes a verb that indicates action, such as OPEN, MOVE, ADD, or PERFORM, is the ______ statement.

imperative

4 The type of statement in the PROCEDURE DIVISION that includes use of the IF or AT END is a ______ expression.

conditional

■ INPUT-OUTPUT VERBS

In this section we consider four input-output verbs: OPEN, READ, WRITE, and CLOSE.

Before an input or an output file can be used by the program, it must be opened. It is not our purpose to go into detail at this point, except to say that opening a file involves checking the availability of the device associated with the file. The following statement can be seen in the first paragraph of the PROCEDURE DIVISION in Figure 2-2:

```
OPEN INPUT    CUSTOMER-FILE
     OUTPUT   REPORT-FILE.
```

The file named CUSTOMER-FILE is opened as input, and the file named REPORT-FILE is opened as output. Thus, the OPEN verb declares the input or output function of the file. It will be recalled that the file names appear in two other divisions of the program. In the ENVIRONMENT DIVISION the two files are referenced in SELECT statements, and in the DATA DIVISION the FD entries name and describe each file. Thus, when references to files are made in the PROCEDURE DIVISION, it is understood that information about the files already has been given in the ENVIRONMENT and DATA divisions.

The basic format of the READ instruction is: READ file-name RECORD AT END imperative statement. In Figure 2-2, in the READ-CUSTOMER-RECORD paragraph, the input instruction is:

```
READ CUSTOMER-FILE RECORD
     AT END MOVE 'YES' TO END-OF-FILE.
```

Each time the command READ CUSTOMER-FILE RECORD is executed, the data contained in that record of the customer file is copied into central storage in the area referenced by the record-name CUSTOMER-RECORD. For example, whatever data is in columns 1–13 of the record is copied in columns 1–13 of

CUSTOMER-RECORD (which part of the record is designated by the data-name CUSTOMER-NAME).

Each execution of the READ statement causes the previous content of the corresponding record in central storage to be erased. This means that normally one record at a time is processed, and, when a next record is read there is no longer any use for the content of the preceding record.

As part of the READ instruction, we also need to indicate what the computer should do after all the input records have been read. The AT END clause serves this purpose. When a record is read, it is examined to see if it is an end-of-file record. The specific form of an end-of-file record differs according to the computer used, but in general it contains data codes that designate it as such. Only when such a record is read is the imperative statement following AT END executed. Thus AT END is a conditional clause: it indicates that the statement following the AT END should be executed if the record just read is an end-of-file record. In our sample program, the imperative statement following AT END enters a YES in the END-OF-FILE field so that the program will be able to determine when all the data have been read in.

The output verb WRITE is similar to the input verb READ, except that reference is made to a record-name rather than a file-name. Consider the following excerpt from Figure 2-2:

```
MOVE REPORT-HEADING TO REPORT-RECORD
WRITE REPORT-RECORD AFTER ADVANCING PAGE.
```

The contents of the WORKING-STORAGE field called REPORT-HEADING are MOVEd (copied) to REPORT-RECORD, which is the record of the output file (OPEN . . . OUTPUT REPORT-FILE).

The AFTER ADVANCING PAGE clause specifies that the printer should skip to the start of a new page and then write the data. PAGE is another reserved word in COBOL and signals the start of a page. As also illustrated in Figure 2-2, we can control the number of lines between printed output. To single space we say WRITE . . . AFTER ADVANCING 1 LINE. To double space we say AFTER ADVANCING 2 LINES, and so on.

As illustrated in Figure 2-4 and also in the program in Figure 2-2, in COBOL we do not write data directly. First we transfer whatever is to be written to an output file record, and then we issue the command WRITE. The output record is like a gate through which all output passes. For that reason, the description of the REPORT-RECORD in the DATA DIVISION specifies one long field of 132 bytes. There is no point in subdividing the field into subfields, since a variety of intended output may be moved to the field. Incidentally, the choice of 132 bytes is in reference to the common width of printers. If our output device were a video monitor or a narrow printer, we would have used another record size, such as PIC X(80).

We conclude this discussion of input-output verbs with the CLOSE verb, which is used after a file is no longer needed and which must be used before the end of the program. At the end of the PROGRAM-SUMMARY paragraph in Figure 2-2, we see:

```
CLOSE CUSTOMER-FILE
      REPORT-FILE.
```

The CLOSE verb is particularly meaningful in the context of magnetic tapes and disk files. File-names are written on separate lines simply to enhance readability.

Review

1 Input of data into the central storage of the computer is accomplished by executing a READ statement. Before a READ statement can be executed, a

checking procedure must be carried out to determine file availability by executing a(n) __________ statement.

OPEN

2 As each record of a file is read into storage, the previous content of that storage location, which typically represents data from the preceding record that was read, is automatically [moved / erased].

erased

3 The part of the READ statement that indicates what should be done after all the records of the input file have been read is the __________ clause.

AT END

4 Output of data from a designated output file is accomplished by executing a(n) __________ instruction.

WRITE

5 Before a WRITE statement can be executed, availability of the output file must be ascertained by executing an appropriate __________ statement.

OPEN

6 An option available with the WRITE statement allows control of the vertical spacing in the printed output. The clause used to designate spacing instruction is the AFTER __________ clause.

ADVANCING

7 When a data processing operation is completed, the availability of both the input and output files that have been used should be terminated. This is accomplished by an appropriate __________ instruction.

CLOSE

THE MOVE INSTRUCTION

The MOVE verb is used to copy data from a sending field to a receiving field. Despite what the word MOVE implies, data in fact are not moved but, rather, are simply copied. Thus, the instruction MOVE A to B designates that B should contain a copy of the content of A. Examples of the MOVE instruction in Figure 2-2 are:

```
MOVE    REPORT-HEADING TO REPORT-RECORD
MOVE    SPACES TO REPORT-RECORD
MOVE    NET-SALES TO REPORT-NET-SALES
```

When numeric data is moved to an edited numeric field, the data in the receiving field is edited to conform with the editing specified in the PIC clause of the receiving field. In the last of the three example MOVE statements above, the PICTURE of NET-SALES is 9999V99 whereas the PICTURE OF REPORT-NET-SALES is ZZZ9.99. If NET-SALES contained 012346, REPORT-NET-SALES would contain 123.46, which includes a blank space before the first digit and a decimal point inserted in the position specified by the PIC clause.

The sending field should never be longer than the receiving field, for otherwise some data would be truncated. When moving a field such as REPORT-HEADING to a larger receiving field such as REPORT-RECORD, the extra spaces to the right are filled with blanks. When moving numeric data, there are many rules to be concerned about. The MOVE operation will be described in further detail in Chapter 5.

Review

1 Execution of a MOVE instruction results in the content of a sending field being [moved / copied] into the receiving field.

copied

2 When numeric data is moved to an edited numeric field, the data is edited as specified in the ________________ clause of the [sending / receiving] field.

PIC; *receiving*

3 The sending field should never be [smaller / larger] than the receiving field.

larger

ARITHMETIC VERBS

In this section we consider the use of four arithmetic verbs: ADD, SUBTRACT, MULTIPLY, and DIVIDE. Our study of these verbs is limited to simple considerations. A more complete presentation is given in Chapter 5, "PROCEDURE DIVISION STATEMENTS."

The ADD verb is illustrated in the sample program in Figure 2-2:

```
ADD CUSTOMER-UNITS TO TOTAL-UNITS
```

The instruction says to add the value in CUSTOMER-UNITS to the one in TOTAL-UNITS. If the first field had a content of 30 and the second a content of 20, the ADD instruction would change the content of TOTAL-UNITS to 50 and would leave CUSTOMER-UNITS unchanged.

TOTAL-UNITS in the above example serves as an *accumulator*. We keep adding to it the values of CUSTOMER-UNITS being read from the CUSTOMER-FILE records. In such a case, it is important to start the accumulator with a zero value, otherwise we would be adding to an already existing value. Therefore, in the DATA DIVISION it is specified:

```
02    TOTAL-UNITS    PIC  9999    VALUE ZERO.
```

The VALUE ZERO clause assures us that the accumulator will have a zero initial value. If we forgot to include the VALUE ZERO clause we would generate improper totals, since the initial content of the field is not likely to be zero.

It should be emphasized that, when a CUSTOMER-UNITS field is defined as numeric (use of the 9 PICTURE character), it does not mean that it will necessarily contain numeric data. A good example is the common error of not filling a numeric field with leading zeros.

Suppose that the CUSTOMER-UNITS PIC 999 field in a data record contained | |2|0|. The first position was left blank. The blank is a distinct character in the memory of a computer, and it is not the same as zero. Execution of ADD CUSTOMER-UNITS TO TOTAL-VALUES would result in erroneous arithmetic, since the addend would not be 20 but "blank 20," which is entirely incorrect.

In light of the above discussion, it also should be obvious now that arithmetic must be performed using fields defined as numeric fields in the DATA DIVISION. What may not be so obvious is that arithmetic must not be performed using numeric edited fields. Thus, fields whose PICTURE clauses contain Z's or editing decimal points (as well as other editing characters to be learned later) must not be operands in arithmetic statements. The 9 and the V, on the other hand, do define numeric items.

Another form of the Add verb is:

```
ADD A B GIVING C
```

which says to form the sum of A and B and store the result in C. A and B are unchanged. This is a convenient way of accomplishing the same result that could be obtained more awkwardly by the several separate commands:

```
MOVE ZERO TO C
ADD A TO C
ADD B TO C
```

Note that in the above program segment the MOVE ZERO command is required to clear C of previous numeric content every time that these instructions are executed. Using the VALUE ZERO clause in the DATA DIVISION would not achieve this objective.

The SUBTRACT, MULTIPLY, and DIVIDE verbs are used in ways that are similar to the use of the ADD. Consider the following examples:

```
SUBTRACT DISCOUNT FROM TOTAL
SUBTRACT 10 FROM TOTAL GIVING NET

MULTIPLY A BY B
MULTIPLY CUSTOMER-UNITS BY PRICE GIVING NET-SALES

DIVIDE PRICE INTO AMOUNT
DIVIDE PRICE INTO AMOUNT GIVING QUANTITY
```

In the first example, TOTAL will be reduced by the value in DISCOUNT. In the second example a constant (10) is subtracted from TOTAL and the result is stored in NET: TOTAL is unchanged. The first multiplication example specifies that the product of A and B is to be stored in B, thus changing B (but leaving the content of A unchanged). The next example is taken from the sample program in Figure 2-2 and illustrates storage of the result in a third field, NET-SALES. Finally, when dividing PRICE INTO AMOUNT the quotient is stored in AMOUNT, while PRICE is unaffected. If we want the result stored in a separate field, the format of the last example is used.

In all cases, we may use numeric literals (numeric constants) instead of data-names, except for the field receiving the result. Thus ADD 10 TO A is correct while ADD A TO 10 is incorrect.

Review

1 Assume that the storage locations with data-names X, Y, and Z contain, both initially and before each statement, the values indicated on the first line of the following table. For each of the COBOL statements in the table, indicate the value contained in each storage location after statement execution. Place a check after any statement that is not an executable ADD instruction.

STATEMENT	VALUE IN X	Y	Z
Initial values:	10	12	100
a ADD X TO Y			
b ADD X Y GIVING Z			
c ADD X Y			
d ADD Z TO Y			

a 10; 22; 100
b 10; 12; 22
c ✓(TO *or* GIVING *is missing*)
d 10; 112; 100

2 Assume that storage locations with data-names X, Y, and Z initially contain the values indicated on the first line of the following table. For each of the table's COBOL statements, indicate the value contained in each storage location after statement execution. Place a check after any statement that is not an executable SUBTRACT instruction.

STATEMENT	X	Y	Z
Initial values:	90	30	20
a SUBTRACT Y FROM Z			
b SUBTRACT Z FROM Y			
c SUBTRACT Z FROM X GIVING Y			
d SUBTRACT X FROM Y GIVING Z			

a 90; 30; −10
b 90; 10; 20
c 90; 70; 20
d 90; 30; −60

3 Assume that the storage locations with data-names X, Y, and Z initially contain the values indicated on the first line of the following table. For each of the table's COBOL statements, indicate the value contained in each storage location after statement execution. Place a check after any statement that is not an executable MULTIPLY or DIVIDE instruction.

STATEMENT	RESULTING VALUE IN X	Y	Z
Initial values:	90	30	10
a MULTIPLY Y BY Z			
b MULTIPLY Y BY Z GIVING X			
c MULTIPLY Z BY Y			
d DIVIDE X Z			
e DIVIDE Z INTO X			
f DIVIDE Z INTO X GIVING Y			

a 90; 30; 300
b 300; 30; 10
c 90; 300; 10
d ✓ *(incorrect form)*
e 9; 30; 10
f 90; 9; 10

CONDITIONAL EXPRESSIONS

Conditional expressions allow the program to test for a condition and to execute one or more commands selectively depending on the result. In the sample program in Figure 2-2 we determine the appropriate price as follows:

```
IF CUSTOMER-UNITS > 20
        MOVE DISCOUNT-PRICE TO PRICE
ELSE
        MOVE FULL-PRICE TO PRICE.
```

The conditional statement begins with the IF followed by a *conditional expression*. In the example, the conditional expression is CUSTOMER-UNITS

> 20. A conditional expression is either true or false. If it is true, the command that follows the IF statement is executed (MOVE DISCOUNT-PRICE TO PRICE); if it is false, the command that follows the ELSE is executed (MOVE FULL-PRICE TO PRICE).

Another example of a conditional statement is:

```
IF MONTH NAME = 'DECEMBER'
    PERFORM YEAR-END-PROCEDURE
ELSE
    MOVE ZERO TO YEAR-TOTAL
    PERFORM MONTHLY-PROCEDURE.
```

In the example above, the conditional expression involves comparisons of a data-name and nonnumeric literal. Further, following the ELSE there are two statements, MOVE and PERFORM, and then there is a period. Both the MOVE and PERFORM statements will be executed if the condition is false (MONTH-NAME is not equal to 'DECEMBER'). As this example illustrates, the period is very important in an IF statement, for it specifies where the conditional statement ends. In order to appreciate the point, suppose that a period were placed after the YEAR-TOTAL data-name in the above example. Then the statement PERFORM MONTHLY-PROCEDURE would be executed *unconditionally*. In other words, no matter what the content of MONTH-NAME, the program would consider the IF statement to end after the MOVE ZERO TO YEAR-TOTAL. By the way, don't be misled by indentation that makes it look like the MOVE and the PERFORM after the ELSE are both dependent on the ELSE. The compiler does not respond to indentation. The first period following the IF is always interpreted as the end of the conditional statement.

Thus far, we have illustrated two *relational operators* in conditional expressions, the greater than (>) and the equal to (=). Following is the complete set of operators that can be used to form conditional expressions.

```
IS GREATER THAN         or     >
IS NOT GREATER THAN     or NOT >
IS LESS THAN            or     <
IS NOT LESS THAN        or NOT <
IS EQUAL TO             or     =
IS NOT EQUAL TO         or NOT =
```

The words and symbols are equally valid, and the choice depends strictly on programmer preference. Thus, the following two expressions are equivalent:

```
IF CUSTOMER-UNITS > 20
IF CUSTOMER-UNITS IS GREATER THAN 20
```

The word IS is optional in all cases, so the following two expressions are both valid and are equivalent to the two expressions above:

```
IF CUSTOMER-UNITS IS > 20
IF CUSTOMER-UNITS GREATER THAN 20
```

Although the conditional IF statement in the sample program includes the ELSE option, it is not necessary to include the option in the command. In the absence of the ELSE option, when the condition tested by the IF statement is true, the command that follows the IF is executed. When it is false, that command is not executed and program execution continues with the next sentence in the program. For example, in the sample program we could have included an IF statement without the ELSE option:

```
IF CUSTOMER-UNITS > 20 MOVE DISCOUNT-PRICE TO PRICE.
MULTIPLY CUSTOMER-UNITS BY PRICE . . . .
```

The above program segment requires that FULL-PRICE is already stored in PRICE before the condition-test is done, perhaps by previous execution of the unconditional command, MOVE FULL-PRICE TO PRICE. Another programming approach by which the differential pricing policy can be achieved without use of the ELSE option is by using two separate IF commands:

```
IF CUSTOMER UNITS > 20 MOVE DISCOUNT-PRICE TO PRICE.
IF CUSTOMER UNITS NOT > 20 MOVE FULL-PRICE TO PRICE.
```

The two alternative programming methods discussed above serve to illustrate the advantage of using the ELSE option to achieve programming simplicity when one of only *two* commands is to be executed based on the result of a condition-test. However, if there are *more* than two alternative commands from which selection is to be made, then multiple IF commands can be used, although nested IF statements, as will be explained in Chapter 6, would be preferred. As an example of using multiple IF commands, suppose that in our sample program a different discount price is to be applied for sales greater than 40 units, in addition to the 20-unit test. Then we could include the following three conditional statements in the program:

```
IF CUSTOMER-UNITS NOT > 20 MOVE FULL-PRICE TO PRICE.
IF CUSTOMER-UNITS > 20 MOVE DISCOUNT-PRICE TO PRICE.
IF CUSTOMER-UNITS > 40 MOVE FULL-DISC-PRICE TO PRICE.
```

Of course, to execute the above program segment the data-name FULL-DISC-PRICE would have been defined in the DATA DIVISION. When CUSTOMER-UNITS in fact exceeds 40 units, note that both the second and third IF statements above would be executed. However, the effect of executing the third IF statement is to replace the content of PRICE resulting from executing the second IF statement with the fully discounted price, FULL-DISC-PRICE.

There are many different ways of forming conditional statements; they will be studied in Chapter 6, "Conditions and Conditional Statements." We conclude this introductory presentation by considering the AT END conditional statement included in the sample program in Figure 2-2.

```
READ CUSTOMER-FILE RECORD
    AT END MOVE 'YES' TO END-OF-FILE.
```

The AT END is a specialized conditional expression that really means "IF END" and refers to the end-of-file record. The specialized AT END conditional statement is used only with the READ verb and specifies what is to be done when our end-of-file type of record is read. In the sample program, we MOVE 'YES' TO END-OF-FILE. Just like any conditional statement, the period is important, although in this particular example it happens that the AT END statement is also the last statement in the paragraph.

Review

1 Whereas the AT END clause is an example of a specialized conditional expression, more general conditional expressions make use of the ________ statement.

IF

2 If the condition specified in the IF clause is met, then program execution continues with [the statement that directly follows the IF / the next sentence / the next paragraph].

the statement that directly follows the IF

3 If the condition specified in the IF clause is not met, the program execution continues with [the statement that directly follows the IF / the next sentence / the next paragraph].

the next sentence

4 The option used in conjunction with the IF clause that makes it possible to execute one of two alternative statements without having to use two separate IF conditionals is the ________________ option.

ELSE

5 Suppose that, if AMOUNT is greater than 10, PRICE should be discounted by 0.20; and if AMOUNT is not greater than 10, PRICE should be discounted by 0.10. Write the COBOL program statement to achieve this result by using the ELSE option.

____

____

____

____

```
IF AMOUNT IS GREATER THAN 10
    MULTIPLY 0.80 BY PRICE
ELSE
    MULTIPLY 0.90 BY PRICE.
```

(*Note that* "MULTIPLY PRICE BY 0.80" *is not correct, because a numeric literal cannot follow* BY *when* GIVING *is not used.)*

6 Using the previous question, accomplish the same computational objective by using two separate IF statements.

____

____

____

____

```
IF AMOUNT IS GREATER THAN 10
    MULTIPLY 0.80 BY PRICE.
IF AMOUNT IS NOT GREATER THAN 10
    MULTIPLY 0.90 BY PRICE.
```

CONTROL VERBS

Program instructions in the PROCEDURE DIVISION are executed in the order in which they are written, from top to bottom, except when control verbs interrupt this normal flow. In this section we describe some basic forms of program control.

The PERFORM verb provides a powerful mechanism for program control. Referring to the PROGRAM-SUMMARY paragraph in Figure 2-2, one can observe the instruction:

```
PERFORM READ-CUSTOMER-RECORD.
```

READ-CUSTOMER-RECORD is a paragraph-name. The effect of the instruction is to branch to the specified paragraph, execute it, and then resume with the statement immediately following the PERFORM. For the specific example in Figure 2-2 we understand that when the PERFORM READ-CUSTOMER-RECORD is executed we read the next record in CUSTOMER-FILE and if that record happens to be the end-of-file record, the field named END-OF-FILE would then contain a YES.

Again, a PERFORM statement says "go execute the named paragraph and come back."

In the sample program in Figure 2-2 we have two PERFORM statements in succession:

```
PERFORM READ-CUSTOMER-RECORD
PERFORM PROCESS-CUSTOMER-DETAIL
    UNTIL END-OF-FILE = 'YES'.
```

The second PERFORM statement has a different format from that of the first statement, as will be explained below. But apart from that consideration, when the program comes to the second PERFORM a record has already been read from the CUSTOMER-FILE including execution of the AT END test in the conditional statement associated with the READ statement.

The PERFORM verb also can be used to execute an iterative procedure, or loop, as in the second PERFORM statement discussed above:

```
PERFORM PROCESS-CUSTOMER-DETAIL
    UNTIL END-OF-FILE = 'YES'.
```

Recall that the END-OF-FILE contains YES only when the end-of-file record has been read in the CUSTOMER-FILE. The above PERFORM operates as follows: the conditional expression END-OF-FILE = 'YES' is tested; if it is true the program leaves the associated PERFORM statement and continues execution with the next statement, which starts with MOVE TOTAL-UNITS. If the condition is not true then PROCESS-CUSTOMER-DETAIL is executed again. The entire process is repeated until the condition is true.

It is important to recognize that the specialized conditional UNTIL . . . is executed first, and if END-OF-FILE is YES then the associated PERFORM-PROCESS-CUSTOMER-DETAIL is not executed. Figure 2-5 is a flowchart that shows this control function.

In order to understand fully the control structure in the sample program, we should notice that the last statement in the PROCESS-CUSTOMER-DETAIL paragraph in Figure 2-2 is a PERFORM READ-CUSTOMER-RECORD. Thus, every time we repeat execution of the PERFORM . . . UNTIL a new record is read and checked for the end-of-file condition. Furthermore, before the PERFORM . . . UNTIL is executed for the very first time, a record has already been read due to the first PERFORM statement in the PROGRAM-SUMMARY paragraph.

Figure 2-6 portrays the control structure in flowchart form, and is an expanded version of the partial flowchart in Figure 2-5. Study this flowchart before you

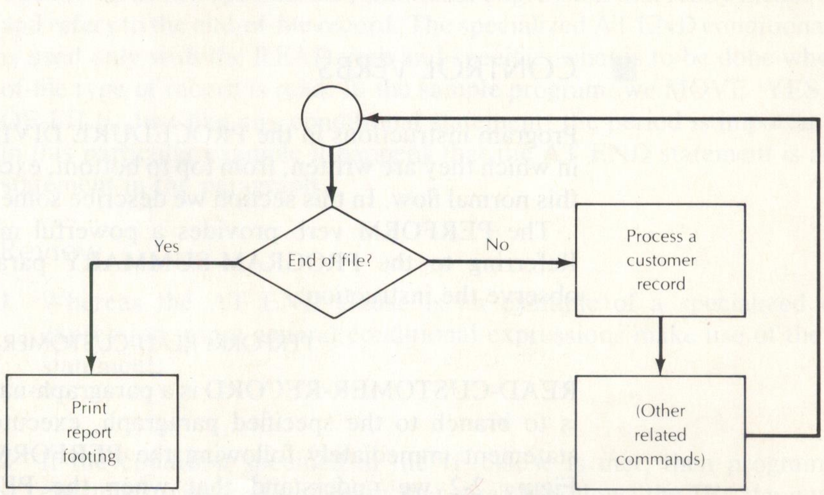


FIGURE 2-5
FLOWCHART SHOWING THE EFFECT OF EXECUTING THE CONDITIONAL UNTIL.

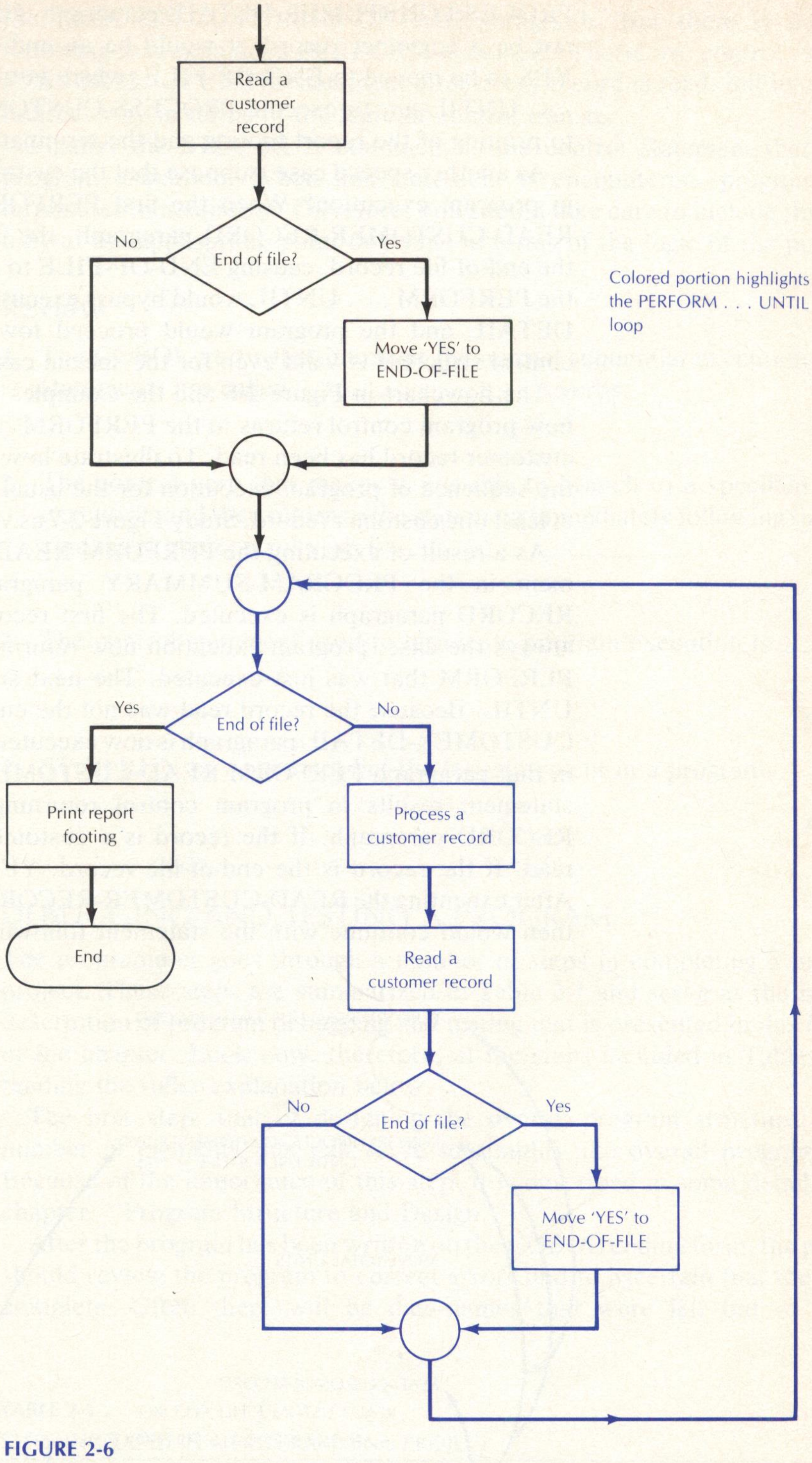


FIGURE 2-6
FLOW OF CONTROL RELATING TO PERFORM STATEMENTS IN THE SAMPLE PROGRAM.

continue reading, including particularly the shaded area that highlights the PERFORM . . . UNTIL loop. As an aid in understanding the control structure involved, we consider a few special cases in the following paragraphs.

Suppose that the customer file contains only two customer records. How many times would the READ-CUSTOMER-RECORD paragraph be executed? The answer is three times. The first time is caused by the PERFORM in the PROGRAM-SUMMARY. The second and third times are caused by the PERFORM in the

PROCESS-CUSTOMER-DETAIL paragraph. Of course, the third record would not be a customer record; it would be an end-of-file record that would cause YES to be moved to END-OF-FILE, which in turn would cause the PERFORM . . . UNTIL *not* to execute PROCESS-CUSTOMER-DETAIL, and would lead to printing of the report footing and the termination of the program.

As another special case, suppose that the customer file is empty. What happens in program execution? When the first PERFORM results in execution of the READ-CUSTOMER-RECORD paragraph, the first record in the file would be the end-of-file record, causing END-OF-FILE to contain a YES. Because of that, the PERFORM . . . UNTIL would bypass execution of PROCESS-CUSTOMER-DETAIL and the program would proceed toward its termination. Thus, the control structure is valid even for the special case of an empty file.

The flowchart in Figure 2-6 and the examples above may not make it obvious how program control returns to the PERFORM . . . UNTIL statement after each customer record has been read. To illustrate how this occurs, Figure 2-7 portrays the sequence of program execution for the usual data situation in which there is at least one customer record. Study Figure 2-7 as you read the following paragraph.

As a result of executing the PERFORM READ-CUSTOMER-RECORD statement in the PROGRAM-SUMMARY paragraph, the READ-CUSTOMER-RECORD paragraph is executed. The first record is a customer record. As is always the case, program execution now returns to the statement following the PERFORM that was just executed. The next statement is the PERFORM . . . UNTIL. Because the record read was not the end-of-file record, the PROCESS-CUSTOMER-DETAIL paragraph is now executed, culminating with the statement in that paragraph PERFORM READ-CUSTOMER-RECORD. Execution of this statement results in program control returning to the READ-CUSTOMER-RECORD paragraph. If the record is a customer record, the record is simply read. If the record is the end-of-file record, YES is moved to END-OF-FILE. After executing the READ-CUSTOMER-RECORD command, program execution then would continue with the statement following the PERFORM statement in

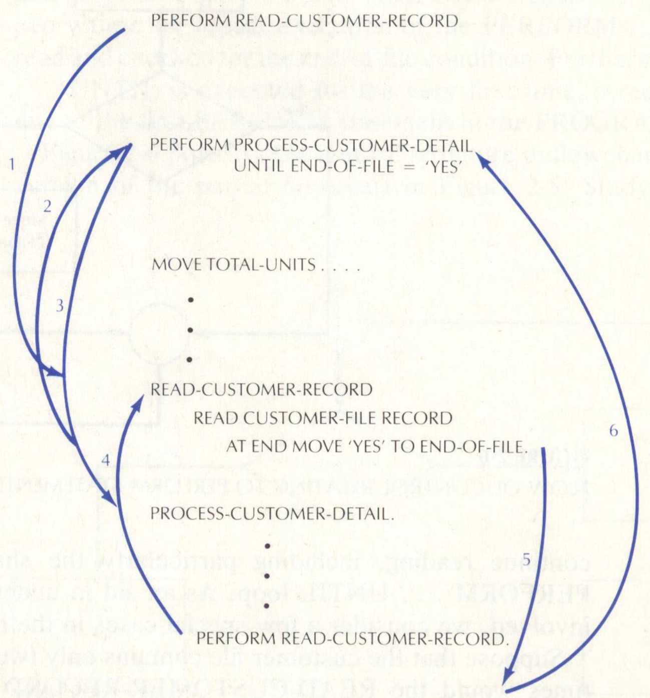


FIGURE 2-7
SEQUENCE OF PROGRAM EXECUTION FOR THE CASE OF ONE CUSTOMER RECORD.

the PROCESS-CUSTOMER-DETAIL paragraph. But there is no statement following that embedded PERFORM statement! Therefore, control returns to the PERFORM . . . UNTIL statement after every record is read, but by a somewhat indirect way in terms of the logic of control transfer.

Finally, the STOP RUN statement is the control statement that terminates program execution. When this statement is encountered, program execution terminates immediately. Therefore, you should take care to include this command only at the appropriate point or points in terms of the logic of the program.

Review

1 The COBOL verbs that interrupt the normal sequential execution of program statements are called ______________ verbs.

control

2 The control verb that makes it possible to branch to a specified paragraph, execute it, and then return to the statement immediately following the statement containing the control verb is ______________.

PERFORM

3 The control statement used to terminate program execution is ______________.

STOP RUN

4 STOP RUN [is / need not be] the last statement in a program.

need not be

DEBUGGING AND TESTING A PROGRAM

The programmer goes through a number of steps in completing a programming project. These steps are summarized in Table 2-1 and serve as the basis for the description of program debugging and testing that is presented in this final section of the chapter. Look now, therefore, at the steps included in Table 2-1, before reading the fuller explanation below.

The first step, that of designing the overall program structure, includes a number of elements that can serve to simplify the overall programming task. Because of the importance of this step, it is described in some detail in the next chapter: "Program Structure and Design."

After the program has been written on the COBOL coding form, the programmer should review the program to correct errors and to ascertain that the program is complete. Often there will be data-names that were left out of the DATA

TABLE 2-1
STEPS INCLUDED IN A PROGRAMMING PROJECT

- Design the overall program structure.
- Write the program on a COBOL coding form.
- Review and correct the handwritten program.
- Key-in the program in machine-readable form.
- Obtain a listing of the program for review.
- Compile the program and review any diagnostic error messages.
- Recompile the program until no error diagnostics are issued.
- Execute the program with some limited test data.
- Review the output for the test data and determine the causes of any erroneous output.

DIVISION and there may be entire paragraphs left out of the PROCEDURE DIVISION. For instance, we may encounter a PERFORM statement that refers to a paragraph-name that we have forgotten to write.

After all apparent errors and omissions in the handwritten program have been corrected, an online terminal typically is used to key-in the program into machine-readable form. The keyed-in version then is listed, allowing for another review in search for any further errors or omissions.

The next step in the general procedure is to compile the program and to obtain a source listing of the program from the COBOL compiler. Such a listing includes *diagnostic error messages* as part of the output. The most common types of diagnostics result from spelling errors or improper syntax, such as missing periods, parentheses, or data-names. The diagnostic messages vary from compiler to compiler, and often require some experience for interpretation of their meaning. Some examples of spelling-related and syntactical errors are included in the following statements:

```
1. . . . LABEL RECORDS OMMITTED
         DATA RECORD IS EMPL-RECORD.
2. 01    EMPL-RECORD
3.       02  EMPL-NAME      PICX(30).
4.       02  EMP-SALARY     PIC 9(4V99.
5. ADD TOTAL-MALES TO 1
6. MOVE EMP-NAME TO EMPL-NAM-OUT
```

The errors in the previous list of statements are:

1 OMMITTED is misspelled; it should be OMITTED.
2 The period was left out after 01 EMPL-RECORD.
3 The space is missing between PIC and X(30).
4 There is no corresponding right parenthesis in 9(4V99; it should be 9(4)V99.
5 The syntax of the ADD statement is incorrect. A data-name must follow TO. The statement should be corrected to ADD 1 TO TOTAL-MALES.
6 EMP-NAME is misspelled. The data name definition in line 3 is EMPL-NAME.

The preceding error diagnostics are *not* of the form that would be obtained from a compiler. The messages received are more general, never tell the programmer how to correct the error, and do not interrelate parts of the program. For instance, in the case of the error given in statement 6, the compiler simply would indicate that EMP-NAME is an undefined data-name.

Based on the diagnostics and review of the program, corrections and changes are made, the program is recompiled, and the process is repeated until no error diagnostics are issued by the compiler. The program then is executed and tested with some limited test data. The data should be representative of the data with which the program eventually will be run but should be limited in the number of data items so that the programmer can determine manually the correct output to be obtained. For instance, for the sample program in this chapter we would include test records with unit sales of zero, greater than zero but less than 20, exactly 20, somewhat greater than 20, and perhaps the maximum permitted value of 999.

If the test run gives any erroneous output, the programmer must find the source of the problem. This step often is the most difficult part of programming. The approach most programmers use is to "desk-check" or "walk-through" the program. In other words, we utilize a listing of the program and the test data and we read the program instructions and apply them to the test data. As we go through the program mentally, we compare our results to the results of program execution, step by step. Often the problem will be somewhere "in the middle" of the program. In such a case, we trace and compare the program execution in

the suspected area of the program, rather than from the very beginning, to save time and avoid unnecessary effort. There may be an early error, however, that only surfaces at a later point in the program. For instance, it may be that a total is incorrect at the end of program execution because we forgot to initialize an accumulator to zero. Thus, if we had forgotten to set the value of TOTAL-UNITS to zero in the sample program, the total units sold and total net sales (in dollars) would be incorrect.

In the sample program in Figure 2-2, the task requires that a line of output be printed as each record is read. In this way we happen to have a natural trace of the input stream. In other programs we may not be printing output as each input record is read; in such a case it often is very useful to modify the program so that we have a trace of the input records. The trace permits us to know whether the program "went bad" as we were processing the first, the 10th, or the last record. Accordingly, we could modify the reading paragraph to list each record on the printer as it is read; for example:

```
READ-CUSTOMER-RECORD.
    READ CUSTOMER-FILE RECORD
        AT END MOVE 'YES' TO END-OF-FILE.
    MOVE CUSTOMER-RECORD TO REPORT-RECORD
    WRITE REPORT-RECORD AFTER ADVANCING 2 LINES.
```

The program now will list every record. Even though the report will be in poor format due to the interspersed printing of report lines and the trace records, we have an easy method of tracing the input.

Of course, we can do a similar trace of any data or computational results. For instance, if we wanted to trace the values of TOTAL-UNITS, we similarly would execute MOVE TOTAL-UNITS TO REPORT-RECORD WRITE REPORT-RECORD each time that we process a record. There actually are alternate ways of accomplishing this all-important trace. Most computer sites have special debugging processors that allow the programmer to specify with ease desired tracings of many or all data- and record-names. Further, COBOL itself contains specialized instructions in its so-called Debugging Module that allow tracing. While both the COBOL and other debugging features are efficient for professional programmers or advanced students, the suggestions just given are an easier way to obtain desired tracing for beginning COBOL programmers.

Experience and situational insight are very useful attributes in debugging and testing; however, the best way to avoid debugging and testing problems is to be very careful in the initial design and writing of the program. It is a tested fact that the student who writes a program with little design and review effort is the one who gets caught in endless attempts to correct a poorly written program. A programmer who spends 10 hours designing, writing, and reviewing a program may spend two hours correcting compiler diagnostics and testing. On the other hand, a programmer with similar abilities who spends just five hours designing, writing, and reviewing for the same program will likely spend 15 hours in debugging the program.

Review

1 After designing the overall program structure, the program is written on a COBOL ________________.

coding form

2 The program on the coding form is reviewed to check for errors and omissions. The program typically is then keyed-in into machine-readable form by means of an online ________________.

terminal

3 After the keyed-in version of the program is listed and reviewed, the program is compiled in order to obtain a source listing and diagnostic ________.

error messages

4 Corrections and changes to the program are made by reviewing all of the diagnostics. When no diagnostic error messages are obtained upon recompiling the program, the program then is executed using limited ________.

test data

5 If the output resulting from the test data is erroneous, the programmer typically desk-checks the program by comparing mental calculations to ________ on a step-by-step basis.

program output

6 When a programmer arranges to list every record or the result of every calculation that is associated with an output error, the procedure is called program ________.

tracing

EXERCISES

2.1 Write COBOL statements for the following computational tasks.

- **a** Set TOTAL equal to the sum of SUBTOTAL-1 and SUBTOTAL-2.
- **b** Add the value of BONUS to PAY.
- **c** Calculate COMMISSION as the product of RATE and SALES.
- **d** Make the value of SHARES equal to the number of shares that can be bought by DOLLARS at PRICE per share.

2.2 Write COBOL statements for the following data transfer tasks.

- **a** Set the value of a data-name A equal to 150.
- **b** Set the value of ACCUMULATOR equal to zero.
- **c** Set HEADER equal to a string of the letter X.

2.3 Write COBOL statements to exchange the contents of two fields, A and B.

2.4 Write COBOL statements to terminate a program if BALANCE is equal to zero.

2.5 Correct the following COBOL program segment:

```
PROCEDURE DIVISION.
FIRST-PARAGRAPH. READ CARD-FILE RECORD AT END (etc.)
```

2.6 Write a statement to make an output file called TRANSACTIONS-FILE available for subsequent use.

2.7 Write statements to determine if an output file called PAYABLES has been read in completely and, if it has, to terminate the program at that point.

2.8 Write statements to print the (last) record of a file called FINAL-REPORT, making sure that the file is closed properly before the program terminates. The record of FINAL-REPORT is called MONTHLY-REPORT.

2.9 Write statements to print two records from a file whose record is called MONTHLY-REPORT, leaving two blank lines between the records. The

first record will print the contents of HEADER-1 and the second will print the contents of HEADER-2.

2.10 Give a program instruction to execute next a paragraph called INTEREST-COMPUTATION.

2.11 If a paragraph consists of four sentences, can we write an instruction outside that paragraph to execute the third sentence next?

2.12 Write instructions to test if the MARITAL-STATUS-CODE is equal to 1 and, if it is, to execute a paragraph called SINGLES.

2.13 A storage field called SEX-CODE indicates whether a person is male or female (1 = male, 2 = female). Another field, OCCUPATION, contains the occupation of the person. Assuming the output record below, write COBOL statements to move the occupation to the appropriate place, making sure that, if a person is male, FEMALE-OCCUPATION is blank, and vice versa.

OTHER-DATA	MALE-OCCUPATION	FEMALE-OCCUPATION

2.14 A student is enrolled in HISTORY, ENGLISH, and ACCOUNTING. These three fields contain his or her corresponding numeric grades in those courses, using this scale: 4 = A; 3 = B; 2 = C; 1 = D; 0 = E. The history course is a two-credit course and the other two are three-credit courses. Write COBOL statements to calculate the grade-point average and store it in GRADE-POINT. Use other storage fields of your choice.

2.15 The GROSS-PAY of an employee is calculated by multiplying REGULAR-HOURS by RATE and OVERTIME-HOURS by 1.50 of RATE. Write COBOL instructions to compute GROSS-PAY, using only one work-area field for intermediate results: WORK-AREA.

2.16 Records are read from a file called CARDS. Each record contains, among other things, a field called AMOUNT. Write COBOL PROCEDURE DIVISION statements to find the smallest value of AMOUNT read in and to store it in SMALLEST. Disregard ties. When all the records have been read, the program continues with the PROCESS-SMALLEST paragraph.

2.17 Is the following PROCEDURE DIVISION statement correct? ADD 3 TO FILLER.

2.18 Assume the following DATA DIVISION entries:

```
01  AMOUNT          PICTURE 9(4).
01  DAYS-ELAPSED    PICTURE X(9).
```

Are the following PROCEDURE DIVISION statements correct?

- **a** MOVE "ZERO" TO AMOUNT.
- **b** MOVE SPACES TO AMOUNT.
- **c** ADD 50 TO DAYS-ELAPSED.
- **d** MOVE ZERO TO DAYS-ELAPSED.
- **e** MOVE "ZERO" TO DAYS-ELAPSED.

2.19 A program contains the statement:

```
PERFORM REPORT-PRINTING UNTIL END-OF-DATA = 'YES'.
```

Explain how END-OF-DATA is defined in the program, and illustrate how it is used as a control mechanism. Assume that END-OF-DATA refers to an input file called CUSTOMER-FILE.

2.20 Using the PERFORM verb, write PROCEDURE DIVISION instructions to read 10 records from a file called CARD-FILE, accumulating the sum of the values in a field called AMOUNT. (*Hint:* Set up a counter to count the records read in and test for the value of the counter to determine termination.)

2.21 Key in and run the sample program given in Figure 2-2. Use as input the sample data shown in Figure 2-1. Even though this exercise will not involve programming effort on your part, it will provide you with the opportunity to familiarize yourself with the ENVIRONMENT DIVISION entries pertinent to your computer, and the program-submission procedures of your installation.

2.22 Write a COBOL program that will take a set of input records and list these records double-spaced on the printer. In other words, each source record read in should be moved to the output file and be written in the exact form that was input.

2.23 Following are the first three divisions of a COBOL program. You are asked to write the PROCEDURE DIVISION for this program and then compile and execute the entire program. The function of this program is to read source records containing the three data fields, NAME, STREET, and CITY, as you can see in the DATA DIVISION under the FD for ADDRESS-FILE. These three fields are located, respectively, in columns 1–25, 26–50, and 51–80 in the source records. The program should print each of these three fields on a separate line, double-spacing between address records.

Output from two sample records would appear as follows:

```
ALLEN M. JOHNSON
1532 E. WASHINGTON ST.
CHICAGO, ILLINOIS 53186

PATRICIA K. WALTON
2252 PALM BOULEVARD
MIAMI, FLORIDA 31322
```

The program can be used with adhesive labels so that a mailing could be made to a group of individuals whose names and addresses are available in ADDRESS-FILE.

The input file, called ADDRESS-FILE, has been ASSIGNed to a device assumed to be called IN-FILE. Obviously this part of the program may have to be changed to adopt it to your own computer. Similarly, the output file, PRINT-FILE, has been ASSIGNed to a device, PRINTER, which may also need revision.

A field in WORKING-STORAGE is called END-OF-DATA-INDICATOR and is initialized to a value of 'NO'. It is assumed that in your PROCEDURE DIVISION you will write:

```
READ ADDRESS-FILE RECORD
    AT END MOVE 'YES' TO END-OF-DATA-INDICATOR
```

and you will PERFORM the repeated function of the program UNTIL END-OF-DATA-INDICATOR = 'YES'.

```
IDENTIFICATION DIVISION.
PROGRAM-ID LABELS.
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT ADDRESS-FILE ASSIGN TO IN-FILE.
    SELECT PRINT-FILE  ASSIGN TO PRINTER.
DATA DIVISION.
FILE SECTION.
FD  ADDRESS-FILE
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS ADDRESS-RECORD.
01  ADDRESS-RECORD.
    02  NAME                        PIC X(25).
    02  STREET                      PIC X(25).
    02  CITY                        PIC X(30).
FD  PRINT-FILE
    LABEL RECORDS ARE OMITTED
    DATA RECORD IS PRINT-LINE.
01  PRINT-LINE                      PIC X(132).
WORKING-STORAGE SECTION.
01  END-OF-DATA-INDICATOR           PIC XXX VALUE 'NO'.
```

2.24 The sample program in Figure 2-2 is to be modified so that the output appears as follows:

CUSTOMER NAME	UNITS SOLD	DISCOUNT	NET SALES
ANDREWS	40	20.00	380.00
JONES	10	0.00	100.00
WILLIAMS	25	12.50	237.50
TOTAL	75	37.50	712.50

In other words, we want to show the amount of the discount given to each customer and the total of all discount amounts. Modify the DATA and PROCEDURE divisions of the sample program to achieve the desired result.

3 PROGRAM STRUCTURE

- INTRODUCTION
- CHARACTERISTICS OF GOOD PROGRAMS
- FLOWCHARTING THE PROGRAM LOGIC
- BASIC FORMS OF STRUCTURED PROGRAMMING
- PROGRAM STRUCTURE CHARTS
- PSEUDOCODE
- AN ILLUSTRATION OF DEVELOPING THE PROGRAM STRUCTURE
- EXERCISES

INTRODUCTION

In Chapter 2 we presented some COBOL language instructions so that we could begin to write complete programs. Knowing the rules of a language is, of course, an essential aspect of the programming process and is the main focus of this book. Still, the development of a program requires more than use of the language. We need to develop a *plan* for the program before we start writing program statements. Therefore, in this chapter we present a number of concepts and tools that are useful in planning or designing a program. These topics are introduced here and are further developed in Chapter 7, ''Program Design.''

CHARACTERISTICS OF GOOD PROGRAMS

The traditional approach to the programming process was to view a computer program as a personal creation by an individual. Often, the characteristic of a good programmer was the ability to write clever programs—''clever'' often being synonymous with complex and obscure. The trouble with such programs is that it is difficult for persons other than the author to understand them, and even the author may have difficulty when months or years intervene. In recent years, therefore, particular emphasis has been given to the principles and concepts by which good programs can be written. The following is a list of several characteristics associated with a good computer program:

1. It is correct.
2. It is understandable.
3. It is easy to change.
4. It has been written efficiently.
5. It executes efficiently.

Of course, the identification of these characteristics does not in itself tell us how to write such programs. However, after identifying these programming objectives, we describe some of the concepts and techniques by which these objectives can be achieved in the subsequent sections of this chapter.

The first, and usually most important, characteristic of a good program is that it is *correct*. The program should carry out the task for which it was designed and do so without error. In order to achieve this objective, a complete and clear specification of the purpose and functions of the program must be developed. Thus, program "errors" can be due to outright mistakes on the part of a programmer or be due to a lack of a clear description regarding the required output of the program.

The second characteristic of a good program is that it is *understandable*. Although a computer program is a set of instructions for a computer, it also should be comprehensible to other people. A person other than the author should be able to read and understand the purpose and functions of the program. Higher-level programming languages such as COBOL are intended for human use and interpretation in their direct form, and are intended for machine use only indirectly, through compilation.

Next, a computer program should be *easy to change*. Changes in products, changes in company procedures, new government regulations, and the like all lead to the necessity of modifying existing computer programs. As a consequence, most established computer installations devote considerable time and effort to changing existing programs. A good program not only fulfills its original purpose but also is adapted easily in response to a changing environment.

The fourth characteristic of a good program is that it has been *written efficiently*. This refers to the amount of time spent in writing the program. Of course, this objective is secondary to the program being correct, understandable, and easy to change. In practice, the easiest way to write a program quickly is to write it partly correct, leave it difficult to understand, or allow its obscurity to make it difficult to change. Still, the principal cost of a programming project is the programmer's time; thus, the best programming techniques economize this time while still satisfying the objectives that the program be correct, understandable, and easy to change.

The final characteristic of a good program that we consider is that it *executes efficiently*. The program should be written so that it does not use more computer storage nor more computer processing time than is necessary. Again, this objective also is secondary to the primary objectives that a program be correct, understandable, and easy to change. Furthermore, as hardware costs have decreased relative to programmers' salaries, overall cost considerations often justify minimizing the concern about a high level of efficiency in program execution. A programmer should be alert, nevertheless, to the techniques by which efficient program execution can be achieved.

Review

1 The most important characteristic of a good computer program is that it is ________________.

correct

2 Probable reference to the program by individuals other than the original programmer dictates that it should be ________________, while unavoidable changes in data processing requirements in most organizations make it desirable that the program be ________________.

understandable; easy to change

3 In order to economize the programmer's time, a program should be ______ efficiently; in order to economize the use of computer hardware, the program should ______________ efficiently.

written; execute

FLOWCHARTING THE PROGRAM LOGIC

A flowchart is a graphic outline of a computer program. As the name suggests, a flowchart identifies the program steps and uses connecting lines with arrows to indicate the flow or sequence of the program steps. Figure 3-1 illustrates some of the most frequently used flowcharting symbols.

Flowcharts are drawn to represent the main logical steps in a program. It would be too time-consuming and unnecessary to draw flowcharting symbols for every statement in a program. If we bothered to do that we might as well write the program code itself. The purpose of a flowchart is to outline the major logical steps in a program and to indicate how they relate to one another in terms of the sequence of program execution. The level of detail that is desirable depends on the circumstances or purpose for drawing a flowchart. We illustrate the point by means of an example.

Figure 3-2 is a reproduction of Figure 2-1 and presents sample output for the programming task discussed in Chapter 2. Since the task is simple, the program logic can be determined just by studying the required output. We see that the whole task can be represented in terms of three main processing steps, or functions, as presented in Figure 3-3*a*. This flowchart is a complete representation of the program, and it may be sufficient for purposes of communicating or analyzing the task. Notice that we have identified the major logical steps and

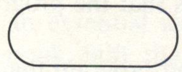
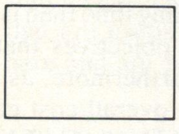
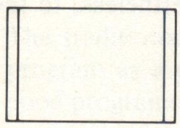
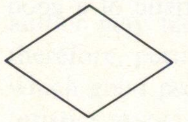
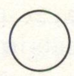
SYMBOL	NAME	MEANING
	Terminal	Beginning or end of a flowchart
	Process	A single step or a series of related steps in a program
	Predefined Process	A multistep process that is shown in greater detail elsewhere
	Decision	A test is performed at this point and then the program takes one of two alternative paths
	Connector	A connecting point between two or more paths

FIGURE 3-1
COMMON FLOWCHARTING SYMBOLS.

CUSTOMER NAME	UNITS SOLD	NET SALES
ADAMS	100	950.00
BROWN	75	712.50
GROVER	30	285.00
MOORE	25	237.50
PETERSON	60	570.00
WILLIAMS	10	100.00
TOTAL	300	2855.00

FIGURE 3-2
SAMPLE SALES REPORT.

their relationships in terms of *sequence*. Although the representation may seem like a trivial accomplishment in this simple task, it often takes a lot of thought and effort before one can break down a programming task into its major parts and establish their time relationships.

Next, we can use the initial flowchart in Figure 3-3*a* as the basis for drawing more detailed representations. In Figure 3-3*b* the second processing step of Figure 3-3*a* has been expanded into a more detailed representation of the program flow. This second version of the flowchart provides a clearer picture of the iterative (loop) process and how control of the process is achieved. We can observe that the specified function is repeatedly executed until the end-of-file condition

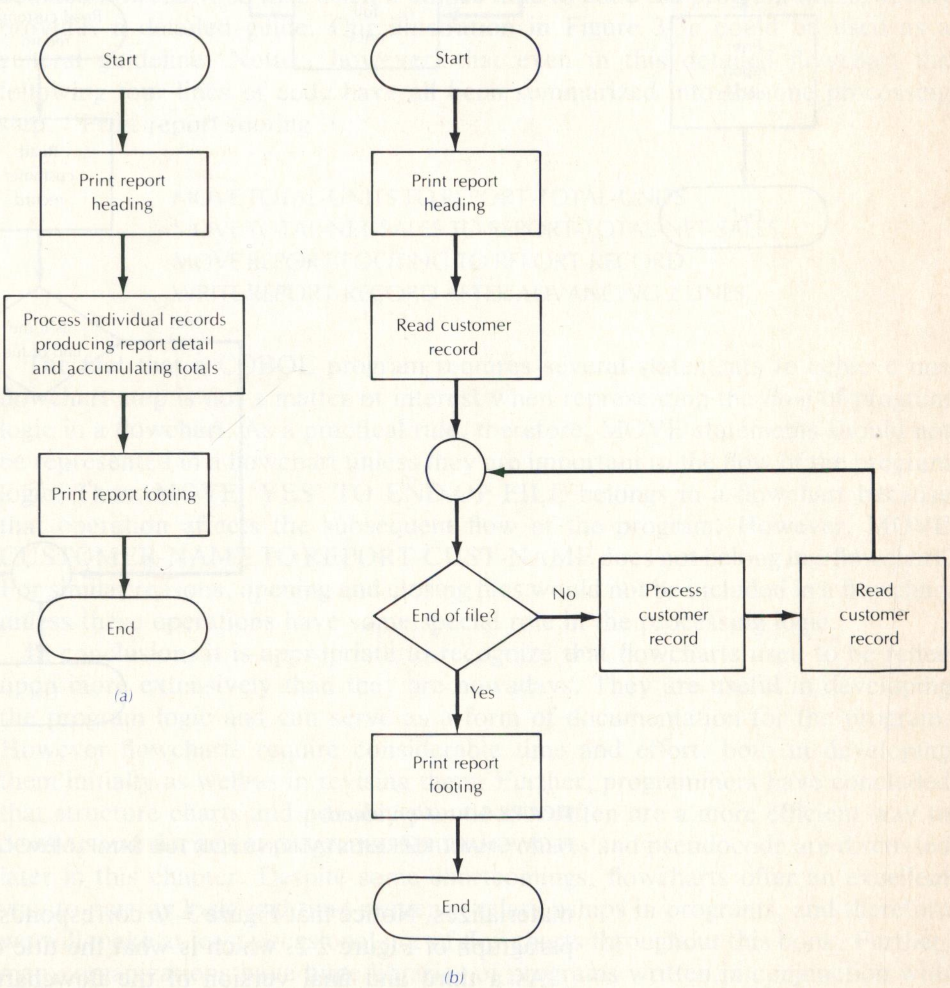
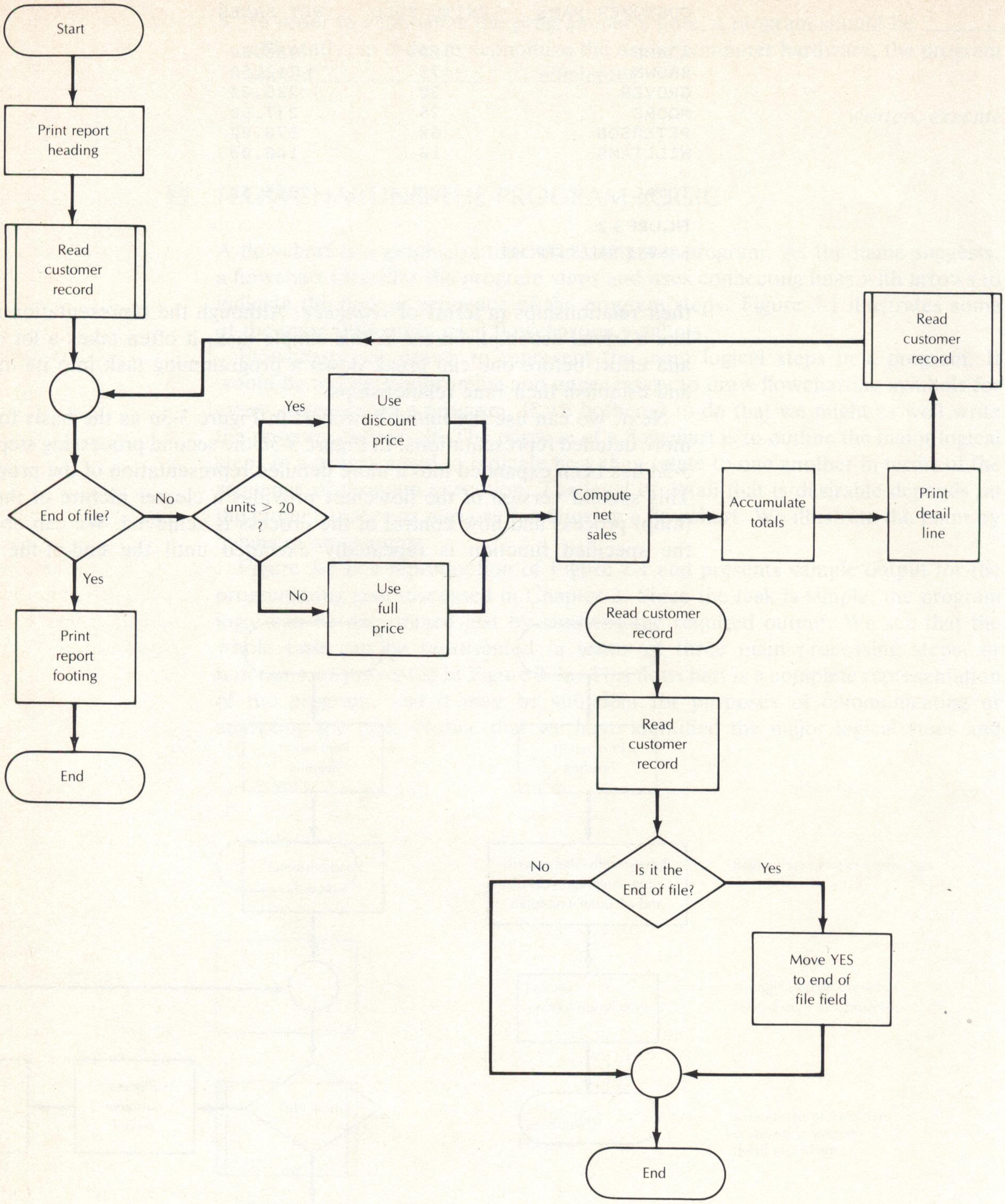


FIGURE 3-3
FLOWCHART REPRESENTATIONS FOR THE SAMPLE PROGRAM.



(c)

FIGURE 3-3 *(Continued)*
FLOWCHART REPRESENTATIONS FOR THE SAMPLE PROGRAM.

materializes. Notice that Figure 3-3*b* corresponds to the PROGRAM-SUMMARY paragraph of Figure 2-2, which is what the title of that paragraph implies.

As a third and final version of the flowchart, Figure 3-3*c* is an even more detailed flowchart, one that corresponds closely to the way the COBOL program is presented in Figure 2-2 in the preceding chapter. This third version of the

flowchart is consistent with the first two versions, but it includes much more specific details about the main program steps.

It is useful to consider some aspects of Figure 3-3*c*. First, notice the use of the "predefined process" symbol in respect to "Read customer record." The reading function is not a single-step process. It involves reading a record *and* testing to determine if it is the end-of-file record. If it is the end of the file, a flag is set to a YES value for use in the subsequent "End of file?" decision step. In addition to being a multistep function, the reading function is specified in two separate places in the flowchart (and eventually in two separate places in the program itself). When we have a multistep function that is specified in more than one place in the program logic, we often set it aside as a separate flowchart segment and refer to it by means of a "predefined process" symbol. By this technique, flowcharts can be kept manageable in terms of complexity. If the same multistep function were repeated in several places, the flowchart would become lengthy, cumbersome, and error-prone.

Our illustration proceeding from a condensed to a detailed flowchart is a recommended approach. It is difficult to do a detailed flowchart from the very start. Instead we proceed from the general to the specific, "blowing up" each portion of the flowchart in greater detail.

The natural question that comes to mind is, at what level of detail do we stop? The answer is, it depends on the purpose of the flowchart and the size and complexity of the task. A beginning student tends to be comfortable with a very detailed flowchart, so that when it comes time to code the program the flowchart provides a detailed guide. Our illustration in Figure 3-3*c* could be used as a general guideline. Notice, however, that even in this detailed flowchart the following four lines of code have all been summarized into the one processing step, "Print report footing":

```
MOVE TOTAL-UNITS TO REPORT-TOTAL-UNITS
MOVE TOTAL-NET-SALES TO REPORT-TOTAL-NET-SALES
MOVE REPORT-FOOTING TO REPORT-RECORD
WRITE REPORT-RECORD AFTER ADVANCING 2 LINES
```

The fact that a COBOL program requires several statements to achieve one flowchart step is not a matter of interest when representing the *flow* of program logic in a flowchart. As a practical rule, therefore, MOVE statements should not be represented in a flowchart unless they are important to the flow of the program logic. Thus, MOVE 'YES' TO END-OF-FILE belongs in a flowchart because that operation affects the subsequent flow of the program. However, MOVE CUSTOMER-NAME TO REPORT-CUST-NAME does not belong in a flowchart. For similar reasons, opening and closing files would not be included in a flowchart unless these operations have some special role in the processing logic.

In conclusion, it is appropriate to recognize that flowcharts used to be relied upon more extensively than they are nowadays. They are useful in developing the program logic and can serve as a form of documentation for the program. However flowcharts require considerable time and effort, both in developing them initially as well as in revising them. Further, programmers have concluded that structure charts and possibly pseudocode often are a more efficient way to develop and document programs. Structure charts and pseudocode are discussed later in this chapter. Despite some shortcomings, flowcharts offer an excellent way to portray logic and time-sequence relationships in programs, and therefore we will make at least occasional use of flowcharts throughout this book. Further, many organizations have huge libraries of programs written in conjunction with flowcharts, and one is likely to encounter a need to understand and use flowcharts in revising previously developed programs.

Review

1 The graphic outline for a computer program that views the task from the standpoint of the sequence of logical steps required to achieve the task is called a ________________.

flowchart

2 Given a particular programming task, the associated flowchart for that task can be expanded by [including additional functions / including more detail for listed functions].

including more detail for listed functions

3 The systematic development of a detailed flowchart following the construction of a first chart involves the process of going from the [general to the specific / specific to the general].

general to the specific

4 In the flowchart in Figure 3-3*c*, the first use of the *terminal* symbol in this chart is for the logical step named "________________."

Start

5 In the flowchart in Figure 3-3*c*, the first use of the *process* symbol is for the step "________________."

Print report heading

6 In the flowchart in Figure 3-3*c*, the first use of the *decision* symbol is for the step "________________."

End of file?

7 In flowcharts, a multistep function that occurs in more than one place in the program logic, and therefore is set aside as a separate flowchart segment, is called a ________________.

predefined process

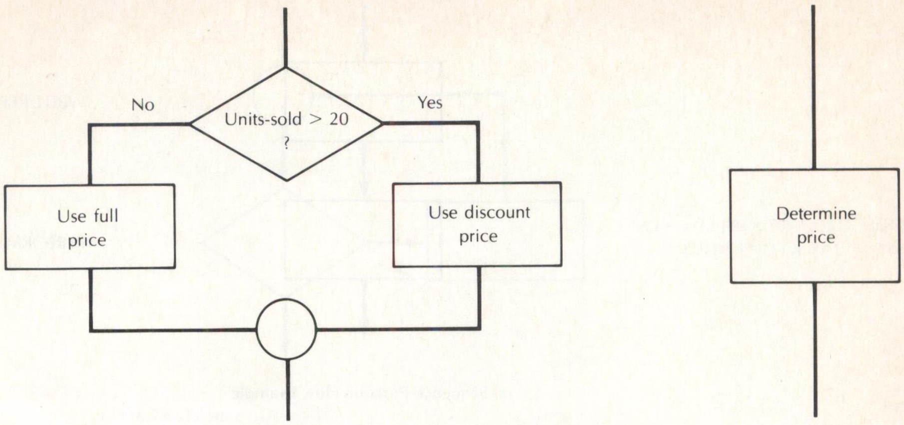
BASIC FORMS OF STRUCTURED PROGRAMMING

In order to write correct, understandable, and easy-to-modify programs, we follow the rules of structured programming which specify that all program flowcharts and program statements be in specified forms. There are five basic forms for structured programs and these are presented in Figure 3-4.

The *sequence* structure consists of a series of flowchart blocks or programming statements. Each item in the sequence may be a detailed step or it may be a whole paragraph. In Figure 3-4*a* the first part of the sequence stands for a detailed step (WRITE . . .), whereas the second step refers to a paragraph that could consist of a few or many steps. In general, in a structured program we are always able to reduce any other form to a sequence. This can be seen by referring back to Figure 3-3. Figure 3-3*a* can be considered as a reduced form of Figure 3-3*c*.

This attribute of structured programs being reducible to sequence forms is very important, because it enables us to simplify those aspects of the task whose complexity is not relevant for present purposes.

For example, consider the following two alternative representations of the same program function:



In the second diagram above, the price determination function is represented as a *black box*; we enter it, we do something, and we exit. At certain times and for certain purposes we don't care to specify how we determine price. A black box is used to represent the process.

It is always possible to implement the black box concept because there is *one entry and one exit* in all of the forms of structure represented in Figure 3-4. Therefore we can always "plug in" a black box representation to stand for any detail between the one entry and one exit.

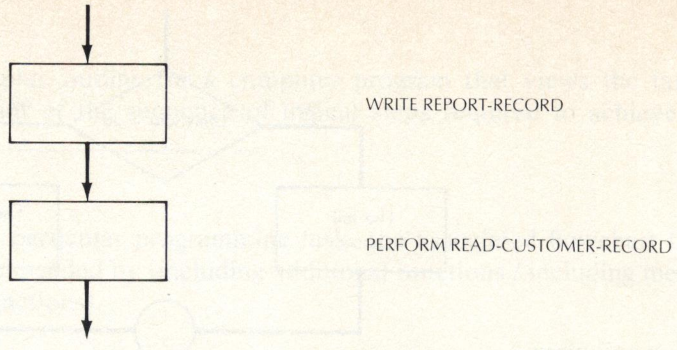
Referring to Figures 3-4*b* and 3-4*c,* we observe two variations of the *decision* form. We can have conditional execution of one process or another in a two-branch decision step, as portrayed in Figure 3-4*b,* or we can have only one conditional branch as portrayed in Figure 3-4*c*. As indicated in Figure 3-4*b,* the two branch decision form is implemented in COBOL by the IF . . . ELSE conditional statement. In the single-branch example in Figure 3-4*c,* if it is the end of the file we do something; if not, program execution continues with the next instruction.

Figures 3-4*d* and 3-4*e* present the two iteration structures In Figure 3-4*d* a test is performed as to whether the indicated block should be executed. Therefore it is possible that there will be no execution of that block. As discussed in Chapter 2, when the customer file is empty, END-OF-FILE is equal to 'YES', and PERFORM PROCESS-CUSTOMER-DETAIL would *not* be executed. As the name *dowhile* implies, this iterative structure is used to specify repeated execution for only as long as some condition holds true.

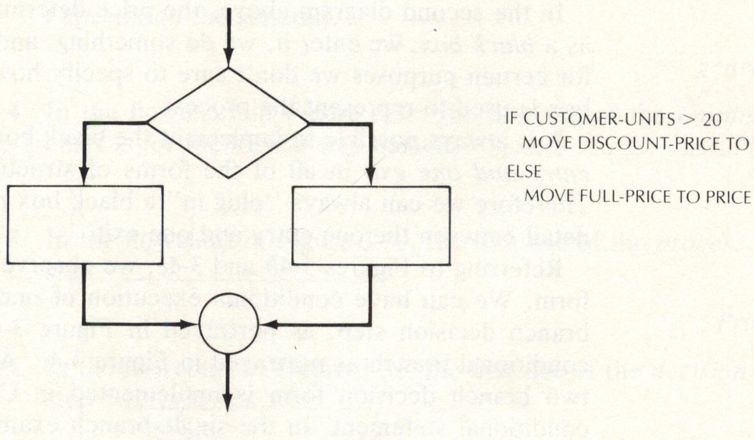
The *post-test* structure in Figure 3-4*e* is used to specify unconditional execution of a task once, and conditional execution thereafter. As the COBOL example in Figure 3-4*e* indicates, we can express this structure by using two statements in sequence (we will also present an alternative procedure in Chapter 5, on PROCEDURE DIVISION statements).

The last structure, portrayed in Figure 3-4*f,* is the *case* form. We use the *case* structure when there is a set of multiple alternative paths in the program logic. This is like a multibranch decision form. Recall that in the typical decision forms we do one alternative or another. In the *case* form we do one alternative, or a second, or a third, and so on. The COBOL implementation of the *case* form will be explained in Chapter 6 on conditional statements, since it requires some additional knowledge of the language. For that reason, Figure 3-4*f* does not include a COBOL example of the *case* structure.

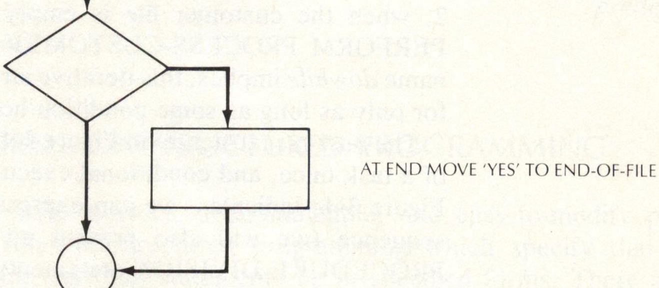
The *sequence, decision, iteration,* and *case* forms of program structure are sufficient to draw any flowchart or write any program. If we were to observe a program structure that does not fit any of these standard forms, then we would conclude that the rules of structured programming have been violated, resulting in undue program complexity and greater likelihood of program error.



(a) Sequence Program Flow Example

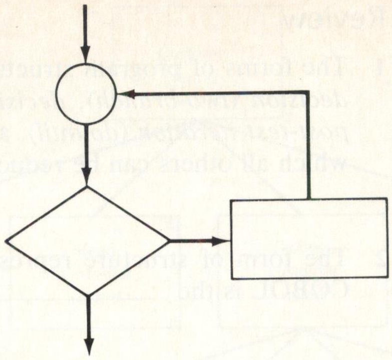


(b) Decision (Two-Branch) Program Flow Example



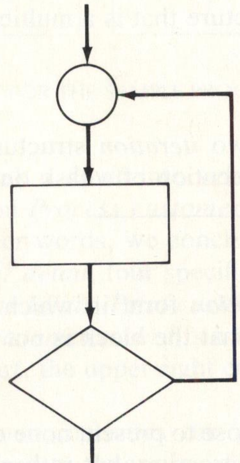
(c) Decision (Single-Branch) Program Flow Example

FIGURE 3-4
FORMS OF PROGRAM STRUCTURE.



PERFORM PROCESS-CUSTOMER-DETAIL
UNTIL END-OF-FILE = 'YES'

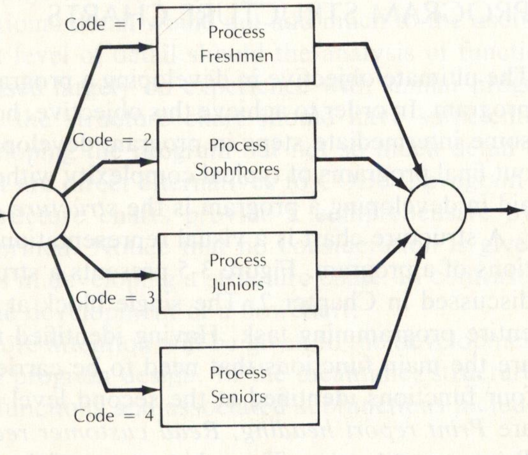
(*d*) Pre-Test Iteration (Dowhile)
Program Flow Example



PERFORM READ-RECORD

PERFORM READ-RECORD
UNTIL END-OF-FILE = 'YES'

(*e*) Post-Test Iteration (Dountil)
Program Flow Example



(COBOL examples are presented in Chapter 6)

(*f*) Case Program Flow Example

FIGURE 3-4 (*Continued*)
FORMS OF PROGRAM STRUCTURE.

Review

1 The forms of program structure considered in this section are the *sequence*, *decision* (*two-branch*), *decision* (*single-branch*), *pretest iteration* (*dowhile*), *post-test iteration* (*dountil*), and the *case* structure. The form of structure to which all others can be reduced is the ________________ form.

sequence

2 The form of structure represented by the use of an AT END statement in COBOL is the ________________.

decision (*single-branch*)

3 The form of structure represented by the use of a conditional IF . . . ELSE statement in COBOL is the ________________.

decision (*two-branch*)

4 The structure that is a multibranch decision form is the ________________ structure.

case

5 Of the two *iteration* structures, the form that is used to specify unconditional execution of a task once, and conditional execution thereafter, is the ________________.

post-test iteration (*dountil*)

6 The *iteration* form in which execution of a block is conditional, and it is possible that the block is not even executed once, is the ________________.

pre-test iteration (*dowhile*)

7 If we choose to present none of the detail associated with a relatively complex task in a program, but rather, represent it in the flowchart by a single block with entry and exit, such a block often is referred as being a "__________."

black box

PROGRAM STRUCTURE CHARTS

The ultimate objective in developing a program, of course, is to write a complete program. In order to achieve this objective, however, it is necessary to go through some intermediate steps in program development. It is just not possible to turn out final programs of typical complexity without programming aids. One valuable aid in developing a program is the *structure chart*.

A structure chart is a visual representation of the main functions and subfunctions of a program. Figure 3-5 presents a structure chart for the sample program discussed in Chapter 2. The single block at the top of the chart stands for the entire programming task. Having identified the overall task we then ask, what are the main functions that need to be carried out in order to do the task? The four functions identified at the second level of the structure chart in Figure 3-5 are *Print report heading, Read customer record, Process customer detail,* and *Print report footing*. The achievement of these four functions would constitute a complete program.

Next, we consider each of the main functions individually and determine what subfunctions need to be carried out to complete that function. For instance, we could ask, what activities need to be carried out to accomplish the function *Print report heading?* Because this task is a simple one, we conclude that it is not

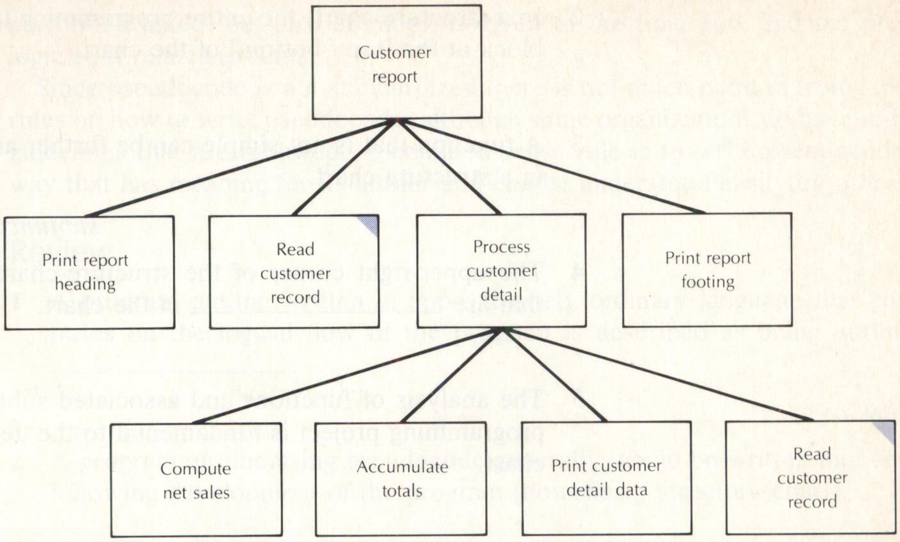


FIGURE 3-5
STRUCTURE CHART FOR THE SAMPLE PROGRAM.

necessary to break it down into *subordinate* functions in Figure 3-5. However, when the function *Process customer detail* is considered, four subfunctions are identified. In other words, we conclude that in order to accomplish the function *Process customer detail,* four specific tasks have to be completed: *Compute net sales, Accumulate totals, Print customer detail data,* and *Read customer record.* Because *Read customer record* is a function that is used more than one place in the structure chart, the upper-right corner of the block is shaded to indicate such repetition.

The programmer continues the process of considering each function and subfunction in turn to determine if it can be broken down into more specific tasks. For example, for the function *Compute net sales* in Figure 3-5 we would ask, what needs to be done to compute net sales? The answer is that the unit price needs to be determined and the units sold are multiplied by the unit price. We could have chosen to show these two tasks as subordinate functions of *Compute net sales* in Figure 3-5. We chose not to do so because showing the additional detail would not add much to the usefulness of the structure chart. At what level of detail should the analysis of functions be stopped? This judgment is based largely on experience with similar programming projects. As a general rule, the structure chart should have sufficient detail to be a useful guide in developing the program but not so much detail that the blocks in the structure chart are direct alternatives to COBOL program statements.

Structure charts provide a comprehensive overview of the functions to be performed. Notice that no consideration is given to the timing or sequence of tasks in developing a structure chart. In contrast, such a concern is fundamental in the development of a flowchart.

More attention will be given to the development of structure charts in Chapter 7 on program design. In the meantime, structure charts will be used to portray the functions and associated subfunctions included in sample programs.

Review

1 The type of chart used as a programming aid that is a visual representation of the main functions and subfunctions of a program is the ______________.

structure chart

2 In a structure chart, the entire programming task is represented by the single block at the [top / bottom] of the chart.

top

3 A function that is not simple can be further analyzed into several ________ in a structure chart.

subfunctions (or subordinate functions)

4 The upper-right corner of the structure-chart block is shaded for functions that are ________________ in the chart.

repeated

5 The analysis of functions and associated subfunctions to be carried out in a programming project is fundamental to the development of a ____________ chart.

structure

6 The graphic outline of the sequence of logical steps required to complete a programming project is the ______________.

flowchart

PSEUDOCODE

After development of a structure chart, it may be useful to write the program outline in *pseudocode*. As the name implies, we do not write the program code according to the rules of COBOL. Instead, we concentrate on the logical flow of the program and we use ordinary language to represent the main functions in the program.

Pseudocode is incomplete and limited, but it is very useful because it can be used to communicate the essence of a program without being bogged down in precise detail. Figure 3-6 illustrates a pseudocode version of the program whose structure chart is presented in Figure 3-5.

Notice the functions presented in the structure chart are cast in a form that approximates a PROCEDURE DIVISION. Not only are the functions to be

```
CUSTOMER REPORT PROGRAM
PROGRAM SUMMARY
    Print the report heading
    Perform Read-Customer-Record
    Perform Process-Customer-Detail until End-of-File = 'YES'
    Print the report footing
    Terminate the program
READ-CUSTOMER-RECORD
    Read a record from the customer file; if it is the special end record then set
      End-of-File to 'YES'.
PROCESS-CUSTOMER-DETAIL
    Determine appropriate price
    Compute net sales
    Accumulate totals
    Print customer detail line
    Perform Read-Customer-Record
```

FIGURE 3-6
PSEUDOCODE FOR THE SAMPLE PROGRAM.

performed listed, but also attention is given to the time flow and the program logic as it relates to time.

Since pseudocode is not standardized there is not much point in trying to offer rules on how to write pseudocode, although some organizations do have in-house guidelines that are followed. A common-sense rule is to write pseudocode in a way that has meaning for its author and can be understood easily by others.

Review

1 A program outline written in approximately ordinary language that concentrates on the logical flow of the program is described as being outlined in __________.

pseudocode

2 A program outline using pseudocode generally would be written immediately following development of the program [flowchart / structure chart].

structure chart

3 In terms of the divisions included in a COBOL program, a program outline in pseudocode approximates program statements that are required in the __________ DIVISION.

PROCEDURE

AN ILLUSTRATION OF DEVELOPING THE PROGRAM STRUCTURE

We conclude this chapter with an example that illustrates the process of developing the structure of a program and includes the use of structure charts, pseudocode, and flowcharts. The use of these aids precedes the writing of the actual COBOL program code.

The sample program task involves reading records from an employee file and producing a report such as the one illustrated in Figure 3-7. Each record in the employee file contains the individual's name, a code classifying the person as a man or woman, and the annual salary. Should there be an omission such that an employee record does not include a sex code, then asterisks are printed in the report to highlight the error. The occurrence of such an error is illustrated in Figure 3-7 for the employee named PROUST, K.

We now proceed to illustrate how structure charts, pseudocode, and flowcharting can be used as aids with respect to this salary-report programming task.

```
                             ANNUAL SALARY

EMPLOYEE NAME                  MEN            WOMEN

JONES, A.                   18,200.00
ANDERSON, P.                12,000.00
ROBERTS, M.                                 15,000.00
PROUST, K.           ******
NICHOLSON, J.               19,600.00
PHILLIPS, P.                                18,500.00
WORK, A.                                    20,000.00


T O T A L                  $49,800.00      $53,500.00

A V E R A G E              $16,600.00      $17,833.33
```

FIGURE 3-7
SAMPLE SALARY REPORT.

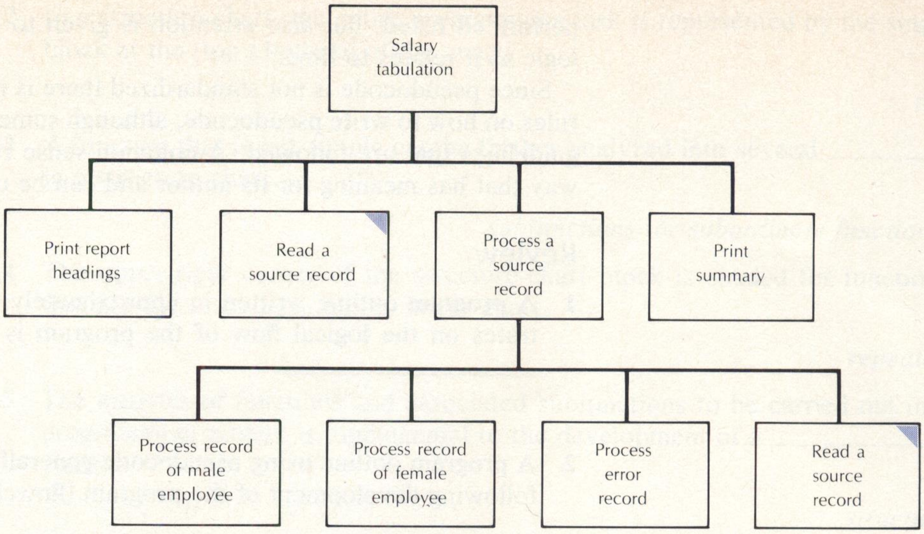


FIGURE 3-8
STRUCTURE CHART FOR THE SALARY REPORT PROGRAM.

Perhaps the best and most natural way to begin designing the program is to compile a list of the main functions to be performed. The main functions for the salary-report program can be listed as:

1 Print the report heading
2 Process each employee record
3 Print the report summary

Further thought about the task leads to development of the structure chart in Figure 3-8. Notice that the *Read a source record* block is shaded in the upper-right corner to highlight the fact that this function is repeated in the chart (in two other places in this case). The *Process a source record* function has been broken down into four more specific subordinate functions.

While Figure 3-8 represents a ''first cut'' at the program design by means of a structure chart, we recognize that the design is not yet finalized. So we think about the task in greater detail and revise the list of functions:

1 Print report heading
2 Process each employee record
  a Read a record
  b Check whether man or woman
  c Accumulate separate totals for men and women
  d Print a line in the report
3 Print the report summary
  a Print the accumulated totals
  b Compute the averages
  c Print the averages

Next we prepare a revised structure chart, as presented in Figure 3-9. Notice again that the Read a Record function is subordinate to two functions. We alert the reader of the chart to this duplication of the function by shading the upper-right-hand corner of the respective blocks. We have included the Read a Record function in the Process a Record function because, in order to process a record for each employee, we need to read the record. It would seem that the Read a Record function could be removed from being a direct subordinate of the top function, Produce Salary Report; however, you would find that if you removed

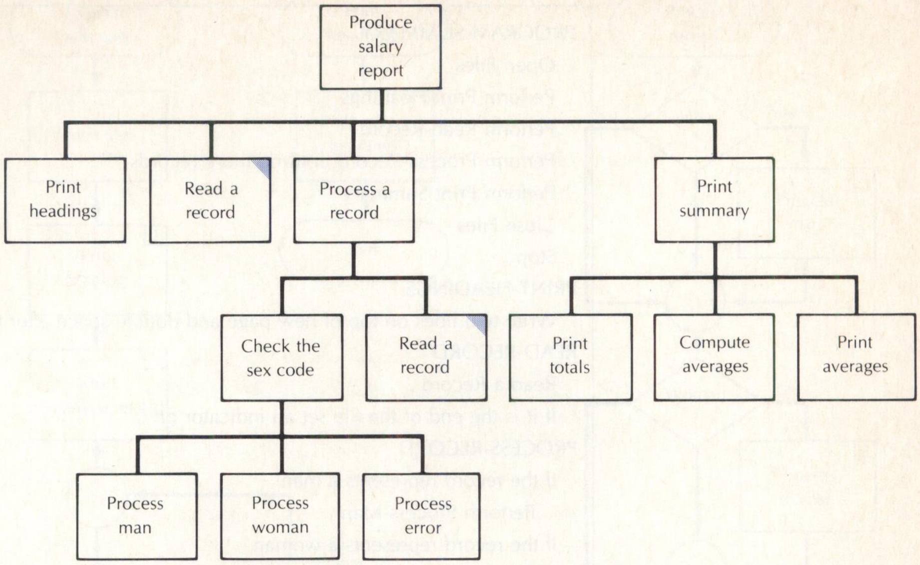


FIGURE 3-9
STRUCTURE CHART FOR THE SALARY REPORT PROGRAM.

this function you would complicate your programming later, even though it seems natural to do so at this point.

Because almost every program reads data from a file and does so repeatedly, it is good to learn the proper approach to structure at this point: design your logic so that you read the first record as a special record and follow this with a loop that processes the record that has just been read and reads the next record as the last thing in the loop. The loop will be repeated until we have reached the end of the file.

We can portray the flow of program execution as shown in Figure 3-10. By this design, the Process Record function will be executed only when there is, in fact, a record to process, and processing stops when the end of the file is reached.

Having made the structure chart for our program, we proceed to develop the program as a series of interrelated procedures. We not only list the functions, but we also specify the order in which they are to be executed. Figure 3-11 presents a program outline using pseudocode. It is written in more-or-less ordinary

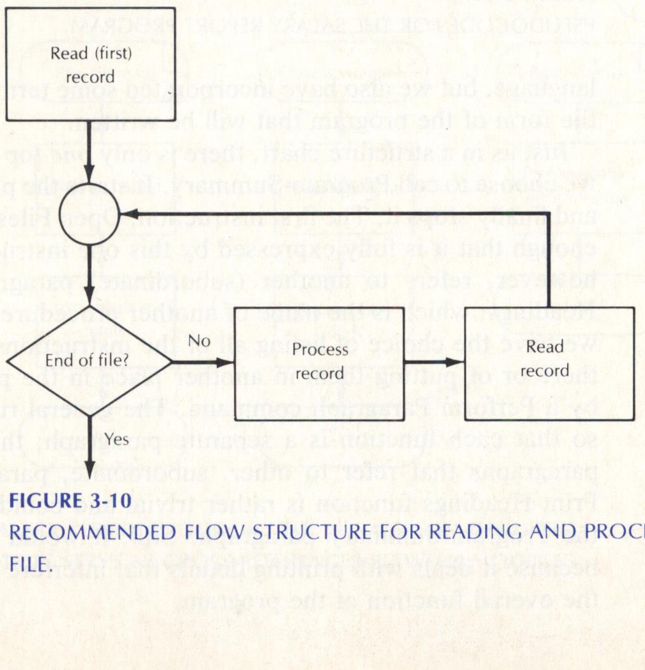


FIGURE 3-10
RECOMMENDED FLOW STRUCTURE FOR READING AND PROCESSING RECORDS FROM A FILE.

```
PROGRAM-SUMMARY
  Open Files
  Perform Print-Headings
  Perform Read-Record
  Perform Process-Record until no more records
  Perform Print Summary
  Close Files
  Stop
PRINT-HEADINGS
  Write two lines on top of new page and double space after the second line
READ-RECORD
  Read a Record
  If it is the end of the file set an indicator on
PROCESS-RECORD
  If the record represents a man
    Perform Process-Man
  If the record represents a woman
    Perform Process-Woman
  If the record represents neither man nor woman
    Perform Process-Error
  Print a report line
  Perform Read-Record
PROCESS-MAN
  Add 1 to number of men
  Add the salary to the total salary for men
  Move the salary data to the men's column
PROCESS-WOMAN
  Add 1 to number of women
  Add the salary to the total salary for women
  Move the salary data to the women's column
PROCESS-ERROR
  Move asterisks to the salary fields
PRINT-SUMMARY
  Print the accumulated total salaries
  Compute the averages (e.g., divide number of men into total salary of men)
  Print the averages
```

FIGURE 3-11
PSEUDOCODE FOR THE SALARY REPORT PROGRAM.

language, but we also have incorporated some terms and functions that anticipate the form of the program that will be written.

Just as in a structure chart, there is only *one* top function in Figure 3-11, which we choose to call Program-Summary. It starts the program, controls its execution, and finally stops it. The first instruction, Open Files, specifies a task that is simple enough that it is fully expressed by this one instruction. The second instruction, however, refers to another (subordinate) paragraph. It says, Perform Print-Headings, which is the name of another procedure. At any point in the program, we have the choice of listing all of the instructions needed to do a subtask right there or of putting them in another place in the program and referring to them by a Perform Paragraph command. The general rule is to partition the program so that each function is a separate paragraph; thus, it is better to write short paragraphs that refer to other, subordinate, paragraphs. In this example, the Print-Headings function is rather trivial and could be included in its entirety in the Program-Summary paragraph. Still, it would be distractive to our reading because it deals with printing details that interfere with our ability to understand the overall function of the program.

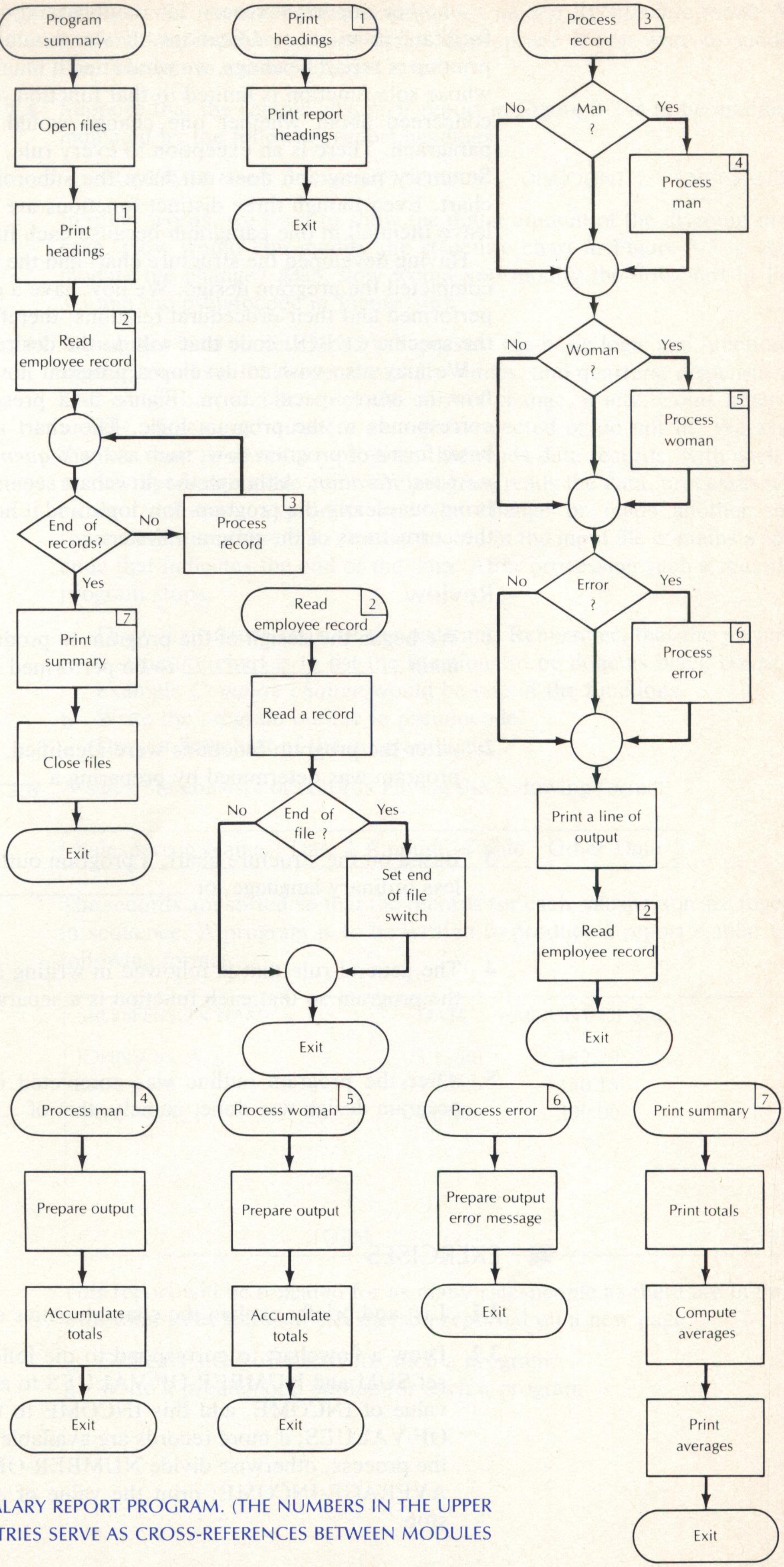


FIGURE 3-12
COMPLETE FLOWCHART FOR THE SALARY REPORT PROGRAM. (THE NUMBERS IN THE UPPER RIGHT-HAND CORNER OF SOME ENTRIES SERVE AS CROSS-REFERENCES BETWEEN MODULES AND THEIR RESPECTIVE BLOCKS.)

Another important reason for modularizing the structure of a program is to facilitate program modifications. If we should want to make a change in the printing of report headings, we would find it much easier to work with a paragraph whose sole function is limited to that function. Otherwise, we would have to be concerned about whether one change would impact something else in that paragraph. There is an exception to every rule, of course. Notice that the Print-Summary paragraph does not have the subordinates identified in the structure chart. Even though three distinct functions are carried out, we have decided to leave them all in one paragraph because each function is so simple.

Having developed the structure chart and the pseudocode, we essentially have completed the program design. We now have a good grasp of the functions to be performed and their procedural relations; therefore, we would be ready to write the specific COBOL code that will do the desired task.

We may also wish to develop a program flowchart to represent the program flow in more specific form. Figure 3-12 presents a complete flowchart that corresponds to the program logic. The chart illustrates the application of the basic forms of program flow, such as the *sequence, decision* (*single-branch*), and *post-test iteration*. Although the flowchart seems to add little in this case, it does bring out clearly the program-flow logic and it helps us to understand and review the correctness of the program better.

Review

1 We began the design of the program to produce a salary report by listing the main ________________ to be performed by the program.

functions

2 After the program functions were identified, the hierarchical structure of the program was determined by preparing a ________________.

structure chart

3 Based on the structure chart, a program outline was prepared using more-or-less ordinary language, or ________________.

pseudocode

4 The general rule that is followed in writing a program outline is to partition the program so that each function is a separate ________________.

paragraph

5 After the program outline was completed in this section, the last step in program design was done, namely, that of ________________.

to write the COBOL *program*

EXERCISES

3.1 List and briefly explain the characteristics of a good program.

3.2 Draw a flowchart to correspond to the following verbal description: start; set SUM and NUMBER-OF-VALUES to zero; input a record containing a value of INCOME, add this INCOME to the SUM, add 1 to NUMBER-OF-VALUES; if more records are available, input another record repeating the process, otherwise divide NUMBER-OF-VALUES into SUM giving an AVERAGE-INCOME; print the value of AVERAGE-INCOME and then stop.

3.3 In Figure 3-5 consider the function block named *Accumulate totals*. What subfunctions would you consider as appropriate if you were to subdivide this function to a more detailed level?

3.4 Suppose that the sample customer report in Figure 3-2 is to be modified as indicated by the following report headings:

CUSTOMER NAME UNITS SOLD DISCOUNT NET SALES

In other words, we want to show the dollar amount of the discount in each case. How would you modify the structure chart in Figure 3-5 to accommodate the change? Also, how would you modify the flowchart in Figure 3-3 and the pseudocode in Figure 3-6?

3.5 A program is to be developed to represent the basic logic and function of a vending machine that accepts nickels, dimes, and quarters, dispenses a 30-cent candy bar, and provides the correct change, if any. Coins other than nickels, dimes, and quarters are either rejected or do not fit. We assume that we want to develop a program that reads data records, with each data record representing one coin. The program reads the data, processes it, and either "delivers" a product and the change or reads another record representing another coin. The last record in the input file contains a special code that indicates the end of the data. After processing such a record, the program stops.

a Draw a structure chart of the program. Remember, that the purpose of a structure chart is to list the functions to be done as black boxes. For example *Compute change* would be one of the functions.
b Write the program outline in pseudocode.
c Draw a flowchart of the program logic.

3.6 A sales file consists of records having the following format:

Salesperson Name	Date	Amount of Sale	Other Data

The records are sorted so that the records for each salesperson are together, in sequence. A program is to be written to produce a report similar to the following format:

SALESPERSON NAME	DATE	AMOUNT OF SALE
JOHNSON, A. J.	03/15/86	132.29
BATES, S. I.	03/20/86	150.15
NOWAK, C. J.	04/01/86	300.00
.	.	.
.	.	.
.	.	.
TOTAL		4,125.89

This report will be repeated for as many salespeople as there are in the file, with total sales for each salesperson reported on a new page.

a Design a structure chart for such a program.
b Write a pseudocode outline for such a program.

4 DATA DIVISION FEATURES

■ INTRODUCTION

As described in the introductory section on the DATA DIVISION in Chapter 2, the purpose of this division of a COBOL program is to identify storage fields and their names. The DATA DIVISION commonly includes two sections: the FILE SECTION and the WORKING-STORAGE SECTION. In this chapter we describe in greater detail the features associated with these two sections of the DATA DIVISION, including the use of various types of PICTURE clauses to define the type of data in storage and the use of several other specialized clauses.

In addition to the FILE SECTION and the WORKING-STORAGE SECTION, the DATA DIVISION also can include a LINKAGE SECTION, REPORT SECTION, and COMMUNICATION SECTION. The LINKAGE SECTION is described in Chapter 15: "Modular Programs and Subroutines." The REPORT SECTION is used with a special feature, the *report writer*. The COMMUNICATION SECTION can be used to access, process, and create messages for communicating with local and remote communication devices. Coverage of the last two sections is outside of the intended scope of this book.

■ DATA STRUCTURE AND RELATIONSHIPS

One of the typical characteristics of data used in administrative applications of the computer is the variety of forms and structure. Figure 4-1 is a reproduction of Figure 1-3 and illustrates the above point. The figure represents a 67-byte record designed to contain customer addresses. A glance at the figure shows that there are many fields, they vary in size, and they are organized into groups. For instance, NAME is a data-name that consists of the combination of the two

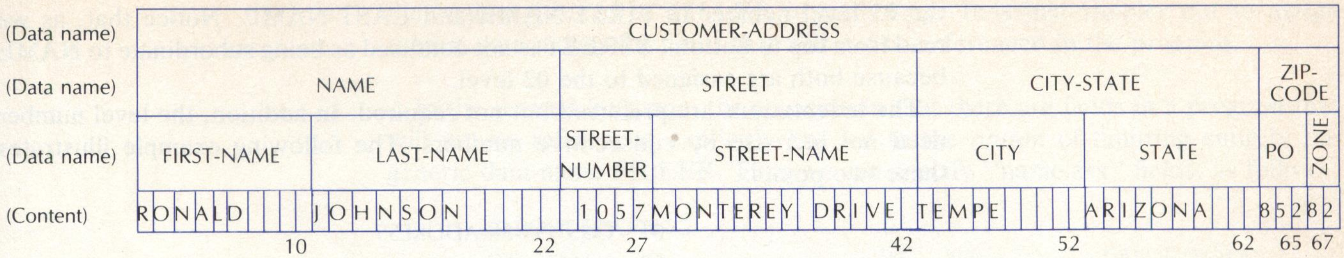


FIGURE 4-1
CONCEPTUAL STRUCTURE OF INFORMATION IN INTERNAL STORAGE.

subordinate fields called FIRST-NAME and LAST-NAME. COBOL derives a great deal of its suitability for business applications from the fact that it allows the programmer to construct hierarchies of data structures.

A useful distinction is made between *elementary* and *group* items in COBOL. An elementary item has no subordinate parts. With reference to Figure 4-1, FIRST-NAME and ZONE exemplify the first and last of the eight elementary items from left to right. The data-name NAME, on the other hand, is an example of a group item. A group item may consist of one or more other group items, as is the case with CUSTOMER-ADDRESS in the illustration.

Reference to Figure 4-1 makes the concept of group item rather obvious, but in a programming language we cannot construct figures, and so we need a means of communicating the same information in symbolic form. COBOL provides such a symbolic form by means of *level numbers*. Here is an example of how level numbers can represent the same hierarchical (grouping) structure as in Figure 4-1:

```
01  CUSTOMER-ADDRESS
    02  NAME
        03  FIRST-NAME
        03  LAST-NAME
    02  STREET
        03  STREET-NUMBER
        03  STREET-NAME
    02  CITY-STATE
        03  CITY
        03  STATE
    02  ZIP-CODE
        03  PO
        03  ZONE
```

The first level number, 01, is associated with CUSTOMER-ADDRESS. An 01 level number indicates the highest level in a data hierarchy. Reference to the data-name at the 01 level is a reference to the entire data set, or *record*, as it is commonly called. There is only one data-name at the 01 level for each record, as it is the all-inclusive data-name. All data-names that follow this one and are part of this record have level numbers that are higher than 01 and, more specifically, are in the allowable range 02–49.

The 02 NAME introduces NAME as a data-name subordinate to the 01 level. Reading from top to bottom corresponds to left to right in Figure 4-1. We observe a total of four data-names at the 02 level: NAME, STREET, CITY-STATE, and ZIP-CODE. Since they are all at the same level, 02, none of them is subordinate to the others in the group (but each is subordinate to the 01 level).

As in Figure 4-1, we are interested in specifying that NAME is a group item and that it consists of two other data-names, FIRST-NAME and LAST-NAME. This relationship is expressed by assigning the 02 level number to NAME and

the 03 level number to FIRST-NAME and LAST-NAME. Notice that, as we read from top to bottom, STREET is not confused as being subordinate to NAME because both are assigned to the 02 level.

The indentations are preferred but not required. In addition, the level number need not increase by consecutive numbers. The following example illustrates these two points:

```
01 CUSTOMER-ADDRESS
03 NAME
05 FIRST-NAME
03 STREET
04 STREET-NUMBER
04 STREET-NAME
```

Notice the absence of indentation in this example and observe that it is much harder to read and understand the expressed data hierarchy than in the preceding example. Also notice that level numbers do not increase by 1. The 03 NAME specifies that NAME is subordinate to CUSTOMER-ADDRESS, because 03 is greater than 01. Similarly, 05 FIRST-NAME is subordinate to NAME, because 05 is greater than 03. In the case of 04 STREET-NUMBER, it is understood that STREET-NUMBER is subordinate to the data-name just above it, which has a lower level number. Thus, the 04 level is perfectly proper in the example, and it preserves the hierarchy of Figure 4-1. It also should be emphasized that, once NAME is assigned to the 03 level, STREET *must* also be assigned to the same level, since NAME and STREET have the same immediate superior, CUSTOMER-ADDRESS.

As illustrated in the above examples, level numbers combine with data-names to represent the desired organization of data. Many times, however, there is no need to describe data in detail. For instance, suppose that we wish to write a program for which the input is the customer file containing the data described in Figure 4-1 and the output is an analysis of the ZIP code information. For such an analysis we are not concerned with the first 62 bytes of data in each record, and therefore we may choose to describe the data record as follows:

```
01 CUSTOMER-ADDRESS.
   02 FILLER      PIC  X(62).
   02 ZIP-CODE.
      02 PO       PIC  999.
      03 ZONE     PIC  99.
```

As explained in Chapter 2, the reserved word FILLER is a generic data-name. It is not unique and therefore cannot be used in the PROCEDURE DIVISION. In the above example its main purpose is to allow us to specify (albeit indirectly) that ZIP-CODE begins with byte 63 in CUSTOMER-ADDRESS.

Because FILLER is used so extensively in programs, we discuss its uses a little further. Another common use of FILLER is in describing report-record formats. For example, in Chapter 2 we saw the following DATA DIVISION entries:

```
01 REPORT-FOOTING.
   02 FILLER                   PIC X(5)      VALUE 'TOTAL'.
   02 FILLER                   PIC X(14)     VALUE SPACES.
   02 REPORT-TOTAL-UNITS       PIC ZZZ9.
   02 FILLER                   PIC X(8)      VALUE SPACES.
   02 REPORT-TOTAL-NET-SALES   PIC ZZZ9.99.
```

Three FILLER descriptions are used to specify the record format for the printed report. The choice of FILLER is not necessary, of course. We could have used

data-names instead of FILLER in each case. It is just simpler not to assign unique data names for fields that will not be referenced in the program.

In the revised language, the recognition that there are fields in a program that will not be referenced explicitly has led to the *option* of omitting entirely the generic data-name FILLER. Thus we can specify "nameless" fields, as follows:

```
01  REPORT-FOOTING.
    02                          PIC  X(5)       VALUE 'TOTAL'.
    02                          PIC  X(14)      VALUE SPACES.
    02  REPORT-TOTAL-UNITS      PIC  ZZZ9.
    02                          PIC  X(8)       VALUE SPACES.
    02  REPORT-TOTAL-NET-SALES  PIC  ZZZ9.99.
```

For entries that contain no data-names, the record description is treated exactly as if FILLER had been specified.

Review

1 In COBOL programming, a data-name that has no subordinate items is called a(n) ______________ item, while one that does have subordinate items is called a(n) ______________ item.

elementary; group

2 For a given data set, the all-inclusive data-name generally would be defined at the numeric ______________ level.

01

3 A data item that is directly subordinate to one at the 02 level [must / need not] be assigned to the 03 level.

need not

4 The generic data-name used extensively in the DATA DIVISION with respect to data items that are not referenced specifically in the PROCEDURE DIVISION is the reserved word ______________.

FILLER

RECORD LAYOUT

As much as COBOL allows great flexibility and ease in describing the organization of data, planning the record layout is a time-consuming and detail-oriented task. When designing a new program or referring to an existing program, it is convenient to use certain graphic representation tools, such as *record layout charts* and *printer spacing charts* to simplify the process.

A record layout chart is intended to help specify and visualize the organization of data within a record. The illustration in Figure 4-1 is certainly capable of serving as a record layout chart, but it is too elaborate for such use and too time consuming to prepare. Instead, we prefer simpler representations, such as the two samples illustrated in Figure 4-2 for the salary-report example presented in the last part of Chapter 3. Figure 4-2*a* is a tablelike representation that is rather easy to scan and understand. Figure 4-2*b* is a more visually oriented chart that corresponds to the left-right concept of data representation. These are simply illustrations. Many organizations use special forms for record layouts. The main point of this brief discussion is that the supporting documentation for a program

COLUMNS	CONTENT	DATA-NAME
1–15	Employee's name	NAME-IN
16	Sex code	SEX-CODE
17–23	Salary	SALARY-IN
24–80	Unused	FILLER

(*a*)

	EMPL-RECORD			
Data-name	NAME-IN	SEX-CODE	SALARY-IN	FILLER
Columns	1 15	16	17 23	24 80

(*b*)

FIGURE 4-2
SAMPLE RECORD LAYOUT REPRESENTATIONS.

should include easy-to-read record layout charts to help convey the organization of data used in the program.

Record layout charts are particularly useful for the type of individual records encountered in using disk and tape files. For report files, however, the focus is on the visualization and description of an entire report rather than single records. Since reports are most often *printed* we refer to such charts as *printer spacing charts*. Actually, their use is equally valid for planning the display of data on a video monitor as well.

Figure 4-3 presents a printer spacing chart for the required salary report illustrated in Figure 3-7 in the preceding chapter. The underlying programming task involves reading records from the employee file, as described in Figure 4-2, and producing the report form specified in Figure 4-3. Such a printer spacing chart allows a visual summary of a report and can be used as a valuable aid in writing the corresponding DATA DIVISION specifications. In order to appreciate the value of a printer spacing chart, we ask two questions:

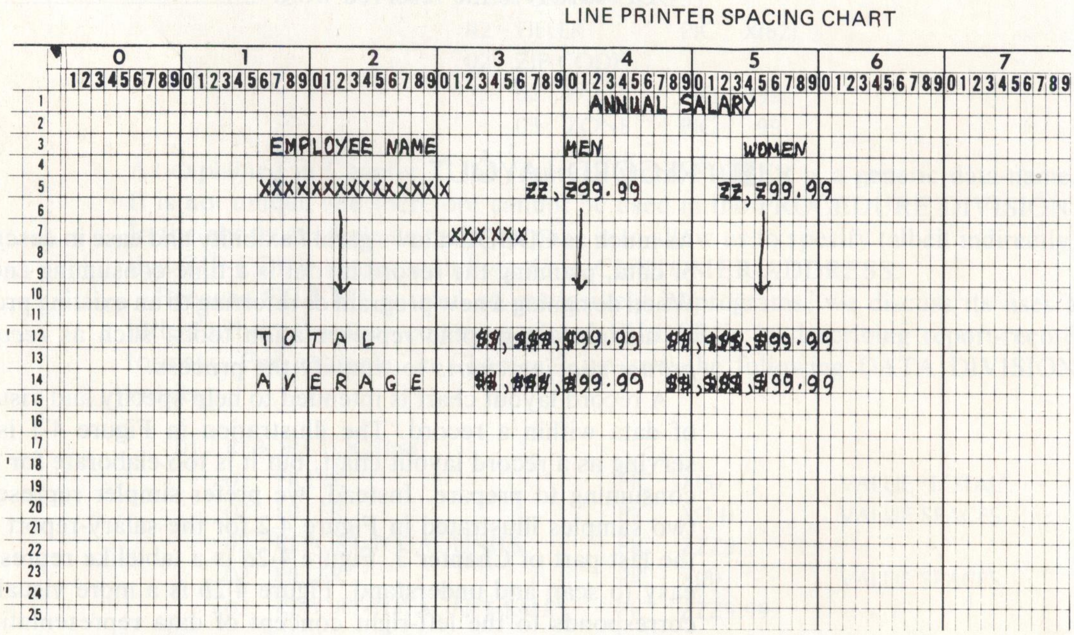


FIGURE 4-3
PRINTER SPACING CHART ILLUSTRATION.

How many record formats are required in the DATA DIVISION for this report output?

How many blank columns will separate the last digit of men's salary from the first digit of women's salary in each report detail line?

In answering the first question, one can observe that there are five different records portrayed in Figure 4-3: lines 1, 3, and 5, and the total and average report lines in the report footing. Finally, in response to the second question, one can observe that there are six blank columns separating the two fields in the printer spacing chart, and therefore an appropriate entry in the DATA DIVISION would be, . . . FILLER PIC X(6) VALUE SPACES.

Review

1 The chart that is used to specify and visualize the organization of data in a *record* is called a ________________ chart.

record layout

2 The chart that is used to portray the layout of a *report,* whether printed or to be presented on a video monitor, is the ________________ chart.

printer spacing

3 The record layout chart and printed spacing chart are particularly useful with respect to writing specifications in the ________________ DIVISION of a COBOL program.

DATA

FILE SECTION

The function of the FILE SECTION is to describe each file used in the program by specifying:

1. The name of the file
2. The name assigned to the record in the file
3. The hierarchical structure of the data fields in the record
4. The field size and type of data in each storage field of the record.

The general format presented thus far in the illustrations in preceding chapters is:

```
FD  file-name
    LABEL RECORDS ARE OMITTED
    RECORD CONTAINS integer CHARACTERS
    DATA RECORD IS record-name.
01  record-name.
```

There is an FD entry for each file involved in a program, which is followed by its record description. The record description begins with the record name, which is always at the 01 level, and includes the data specifications for the data fields in the record. For instance, in the sample program in Chapter 2 we had:

```
*
DATA DIVISION.
*
FILE SECTION.
*
FD  CUSTOMER-FILE
    LABEL RECORDS ARE OMITTED
```

```
        RECORD CONTAINS 80 CHARACTERS
        DATA RECORD IS CUSTOMER-RECORD.
    01  CUSTOMER-RECORD.
        02 CUSTOMER-NAME      PIC  X(13).
        02 CUSTOMER-UNITS     PIC  999.
        02 FILLER             PIC  X(64).
```

The LABEL clause will be discussed in Chapter 9, "Sequential File Processing." For now we simply use the LABEL clause as shown. The RECORD CONTAINS clause is used for documentation. The size of the record is specified indirectly by the sum of the individual PICTURE descriptions, which should correspond to the number of characters identified in the RECORD CONTAINS clause. For the above example the sum of the individual PICTURE descriptions is 13 + 3 + 64 = 80, which does so correspond. Then the DATA RECORD IS specifies the name of the record, which is described beginning with the 01 level number.

In the revised language, the LABEL, RECORD and DATA clauses in the FD entry are all optional. Therefore we could have written:

```
FD  CUSTOMER-FILE.
01  CUSTOMER-RECORD.
    02 . . . (etc.)
```

The reasons for making those clauses optional are as follows: The LABEL aspect of a file in fact generally is dependent on the operating system rather than on COBOL specifications, even in using the 1974 version of the language. The RECORD CONTAINS has always been optional, and the DATA RECORD IS is redundant, since the subsequent 01 entry always specifies the record-name in any event.

Review

1 The section of the DATA DIVISION concerned with describing each file used in the program is the ________ SECTION.

FILE

2 In the FILE SECTION, we always begin a file description by identifying the name of the file in the entry, which is designated by the COBOL reserved word ________.

FD (*for File Descriptor*)

3 After the name of the file is given, the next item of information given in the FD entry is the name of the ________ included in the file.

record

■ WORKING-STORAGE SECTION

The WORKING-STORAGE SECTION immediately follows the FILE SECTION in the program. Although it is possible for a program not to have either a FILE SECTION or a WORKING-STORAGE SECTION, such a program would be very unusual. Whereas the FILE SECTION describes the files used in the program, the WORKING-STORAGE SECTION provides for the storage of data items that are not part of any file, such as intermediate calculations, report headings for printing, and numeric constants for use in calculations.

The WORKING-STORAGE SECTION consists of group items and elementary items, as is also the case in the FILE SECTION. In older versions of COBOL

it was common to use a special level number, 77, to describe all elementary independent items; that is, elementary items that are not part of a group. These elementary independent items preceded all the group (or record) items which were introduced at the 01 level. However, whether the elementary independent items are designated by the level 77 or 01 makes no difference in program processing, and this is the main reason why the use of level 77 was abandoned in the late 1970s.

There is little to say about the organization of the WORKING-STORAGE SECTION except that it is a recommended practice to group related items together under a common superior in order to enhance reader awareness of their relationships. For example, consider the following DATA DIVISION entries:

```
01  SALARY-TOTALS.
    02  MEN-TOTAL-SALARY       PIC 9(7)V99 VALUE ZERO.
    02  WOMEN-TOTAL-SALARY     PIC 9(7)V99 VALUE ZERO.
```

Assuming that we never make use of the data-name SALARY-TOTALS in the program, the 01 group item in this program segment is used only for readability. An alternative to this structure would be to write the two elementary independent items as follows:

```
01  MEN-TOTAL-SALARY       PIC 9(7)V99 VALUE ZERO.
01  WOMEN-TOTAL-SALARY     PIC 9(7)V99 VALUE ZERO.
```

In this alternative program segment, the similarity in the data-names and their physical consecutive order implies a relationship between them, but the association is much clearer when a group data-name is used to indicate the relationship between the two data-items.

Review

1 The section of the DATA DIVISION that provides for the storage of such data items as intermediate calculations and report headings is the __________ SECTION.

WORKING-STORAGE

2 In lieu of the older practice of using the special level number, 77, for elementary independent data items, current practice favors use of the level number __________ for such items.

01

3 When two or more elementary items are related, rather than place them in the program as elementary independent items, documentation is improved by __________.

grouping them under a common superior

THE PICTURE CLAUSE FOR DATA DESCRIPTION

The general form of the PICTURE clause was introduced in Chapter 2 in the example of a complete program. The purpose of this DATA DIVISION feature is to describe the data included in the data items. We use the word PICTURE, or its abbreviated form PIC, and the optional word IS, followed by a string of characters that describe the data. In the following subsections, we consider in turn each of the PICTURE characters that can be used to describe data fields that are defined in the DATA DIVISION.

The 9 PICTURE Character

The numeric 9 indicates that a storage position should only contain any of the numeric digits from 0 to 9. In this context, a blank is *not* considered equivalent to the numeric 0 and thus is not a numeric character. The field size of the item is indicated by the number of successively written 9s in the PICTURE clause; thus, PICTURE IS 999 means a field of three numeric positions. As indicated in Chapter 2, an alternative is to write a 9 followed by parentheses enclosing the number of positions in the field. For example, the statement 03 AMT PICTURE IS 9(5) indicates that AMT is a five-position numeric field. Some examples of using the 9 PICTURE character are as follows:

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 SCHOOL-ENROLLMENT PICTURE IS 9(6)	12,327	012327
02 STOCK-ON-HAND PICTURE 9999	8,956	8956
04 POPULATION-OF-CITY PICTURE 9(10)	1,563,813	0001563813
03 UNION-MEMBERSHIP PICTURE IS 9999	285	0285

Again, note that a numeric field can contain only the digits 0–9. Blanks are *not* numeric characters. When entering data, you should be careful to zero-fill a field with leading zeros; otherwise you may be in for some surprising results. Thus, in a field of six positions, the numeric value 532 should be entered as follows:

0	0	0	5	3	2

The V PICTURE Character

The V character indicates the position of an assumed decimal point. "Assumed" means that the decimal point is not written as part of the field and therefore is not included as part of the field size. Instead, the information about decimal-point location is stored elsewhere in the computer, so that any arithmetic computations can be done correctly. For example, if two items are multiplied and each is assumed to have two positions to the right of the decimal, the product will be understood to have four positions to the right of the decimal point. If the V character is omitted, it is understood that the decimal point is at the extreme right of the numeric field. It is not necessary, therefore, to place a V as the last character in a PICTURE clause. Of course, no more than one V is permitted in a field. Refer to the following table and note that, if we printed HOURS-WORKED without any editing, the value represented in storage, 385, would be printed *without a decimal*. Again, the V character establishes the *position* of the decimal for purposes of arithmetic manipulation but does not make the decimal point as such available for printout. Some examples of using the V character are given in this table. The caret (‸) indicates the position of the assumed decimal point.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
03 HOURS-WORKED PICTURE 99V9	38.50	38‸5
03 NET-PAY PICTURE 9(4)V99	452.39	0452‸39
02 TON-CAPACITY PICTURE 999	550	550‸
02 BALANCE PICTURE 99999V99	23561.00	23561‸00

The P PICTURE Character

The P PICTURE character is used in conjunction with the V character to indicate the position of a decimal point in cases in which the decimal point is not within the number. This character is used, for example, when it is understood that a

value held in storage represents thousands of units and we wish to indicate the decimal position for this value. The following examples indicate the use of this character. As before, the caret indicates the position of an assumed decimal point.

DESCRIPTION	NUMERIC VALUE	ARITHMETIC EQUIVALENT
02 AMOUNT PICTURE 99PPPV	12	12000‸
02 AMOUNT PICTURE VP(3)9(4)	1023	‸0001023

The P character is not used very much in administrative applications. It is suited best to scientific computational needs, which are likely to be better satisfied by the use of languages other than COBOL.

The S PICTURE Character

The S character is used to designate a numeric field that is signed; i.e., one that can be negative in value. In COBOL, all fields are considered positive unless the S has been used. For instance, for a field containing the checking account balance of a bank customer, when the account is overdrawn the only way the balance will become negative is to designate the balance field as a signed one by use of the S character. Otherwise, if the balance is $23.50 and a check is written for $50.00, the balance will become $26.50!

Only one S character may be used in a field, and it must be the leftmost character. The S is not counted in the size of the field, and therefore S99 is a field of two positions. In the following examples, the negative sign in machine representation is shown as a "−" on top of the rightmost digit, in order to preserve the concept that it does not take up an extra position.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 BALANCE PICTURE S9999V99	156.29	0156‸29
02 BALANCE PICTURE S9999V99	−1251.16	1251‸1$\bar{6}$
02 BALANCE PICTURE S9(4)V99	−0.10	0000‸1$\bar{0}$
02 BALANCE PICTURE 9(4)V99	−325.18	0325‸18

It should be understood that we are referring to sign representation in terms of internal storage. In order to represent negative data in a file or at a terminal, other considerations apply. For example, a value of negative 32 might be written as 3K, the K representing the bit combination of the digit 2 *and* the negative sign. Further, the specific representation would depend on the system in use. If this seems a bit alarming, keep in mind that business processing data are almost never negative. Instead, we define a category of positive data that may then be treated as negative inside the machine. Consider this example: A customer returns an item and receives a refund. One could think of this as a negative sale represented by a negative number. But that is not the way it is done. Instead, a "sales" record would have a field that identifies the type of sale, as perhaps purchase or return, by means of a code. When the record is processed we would specify something like this:

If it is a *purchase*
    add the amount to the customer's balance
else
    if it is a *return*
        subtract the amount from the customer's balance.

As the above example illustrates, negative values are not used directly in input files. When we want to show negative values in reports, we use the special editing

PICTURE character "–" as explained in the next section of this chapter, on the use of the PICTURE clause for data editing.

The X PICTURE Character

The X PICTURE character denotes that alphanumeric positions are contained in a field. NAME X(20), for example, signifies a field of 20 alphanumeric positions, which can include alphabetic characters, numeric characters, and special symbols. In the following examples, a "b" represents a blank space in the storage location. Notice that when the characters do not fill an X field completely they are left-justified, with blanks filling the remaining positions on the right.

DESCRIPTION	EXAMPLE	REPRESENTED IN STORAGE AS
02 PART-NAME PICTURE XXXXX	DIODE	DIODE
02 PART-NAME PICTURE X(5)	TUBE	TUBEb
02 NAME PICTURE X(20)	JOHN F. ANDREWS	JOHNbF.bANDREWSbbbbb
02 MESSAGE-CODE PICTURE X(8)	AB13C,$M	AB13C,$M

The A PICTURE Character

The A PICTURE character is similar to the X character, except that it indicates that only alphabetic characters and blanks are contained in a field. Excluded therefore are numeric characters and special symbols. Since the first two statements in the preceding examples concerned storage locations containing only alphabetic information, the A PICTURE character could have been used instead of the X, as indicated in the following table:

DESCRIPTION	EXAMPLE	REPRESENTED IN STORAGE AS
02 PART-NAME PICTURE AAAAA	DIODE	DIODE
02 PART-NAME PICTURE A(5)	TUBE	TUBEb

The reader is cautioned against using the A character in what seems a natural use: a field containing people's names. Names such as O'Neal do not consist of alphabetic characters alone. The X character is better suited for use in such fields.

Review

1 We have discussed thus far six PICTURE characters that can be used in a PICTURE clause to describe the contents of a field. These are the 9, V, P, S, X, and A PICTURE characters. The character used to indicate that a field can contain either alphabetic, numeric, or special symbols is the ________ character; whereas the character that indicates alphabetic content only is the ________ character.

X; A

2 The character that indicates numeric content only is the ________ character. If the stored values can be negative as well as positive, the PICTURE clause should include the ________ character as the leftmost character.

9; S

3 In entering data, a negative value is indicated by including a negative sign in the column containing the [leftmost / rightmost] digit of the field.

rightmost

4 The character used to indicate the position of an assumed decimal point is the ________________ character. This character is used only with PICTURE clauses that also contain the ________________ PICTURE character.

V; 9

5 When we wish to identify the correct decimal position for a field whose numeric value is understood to be in thousands of units, we use the ________________ PICTURE character.

P

6 When a value does not fill a numeric 9 field completely, the value is justified to the [right / left], and the extra positions are filled with [blanks / zeros].

right; zeros

7 When an item does not fill an A field or an X field completely, the item is justified to the [right / left], and the extra positions are filled with [blanks / zeros].

left; blanks

THE PICTURE CLAUSE FOR DATA EDITING

As contrasted to the field definition characters we have described so far, the PICTURE characters that follow are editing symbols. The editing function involves a change in the form of data. For example, we may suppress leading zeros, we may use commas to make long numeric values more legible, we may insert a dollar sign in front of a value, and so forth. The purpose of editing is to make data more suitable for human reading. Thus, in its most common use, editing is associated with printing data on the printer. A great many of the applications of COBOL involve the production of reports that are to be read by people, and data editing greatly enhances the visibility of data in such reports.

The $ PICTURE Insertion Character

By use of the $ PICTURE character, the dollar sign is written in the position in which it is to appear in the output. Since the $ sign is counted in the size of the field, the field should be assigned at least one more position than the maximum number of significant digits expected. The $ sign also may be floated, by which we mean that it will not necessarily be entered in the leftmost position of a field but, rather, will be entered to the left of the first significant digit in the field and be preceded by blanks. For example, if we have the statement 02 AMT PICTURE $$$99V99, when a data value is to be entered in AMT, a test is performed. The leftmost digit is examined first. If it is zero, the next digit is examined. If this next digit is not zero, then the dollar sign is inserted directly to the left of it. For the PICTURE clause above, the $ sign can appear in any one of the first three positions, according to the value stored in the field. The following examples further illustrate the use of the $ PICTURE insertion character. The last example shows that, when the $ sign appears in all positions and the value is zero, the effect is to blank the field.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 AMT PICTURE $999V99	125.13	$125^13
02 AMT PICTURE $9(5)V99	100.0	$00100^00
02 AMT PICTURE $$99V99	12.49	b$12^49
02 AMT PICTURE $$$9V99	150.10	$150^10
02 AMT PICTURE $$$$V99	0.15	bbb$^15
02 AMT PICTURE $$$$V$$	0.0	bbbb^bb

For the last example just given, let us consider what the result in storage would be if the value to be written in AMT were 0.05. In this case, the presence of the V would stop the float, and the value would be represented in storage as bbb$^05. If the decimal point did not terminate the float, the result in storage would be bbbbb^$5, which is clearly not the desired representation. Therefore, the presence of the decimal point stops the float, *except* when the entire field is zero. Further examples involving use of the $ PICTURE character are included in the discussion immediately following.

The Decimal and the Comma PICTURE Insertion Characters

Each of these insertion characters is used to indicate the position of the indicated character in the storage location. Because the . (decimal) PICTURE character indicates the position of the decimal point and serves to align the actual decimal values in the field, only one such character may appear in a field. Further, a field cannot contain both a V and a . PICTURE character. On the other hand, a field may include more than one , (comma) PICTURE character if the size of the field warrants it. The following examples illustrate the use of the . and the , PICTURE insertion characters in conjunction with the $ insertion characters. Notice some of these points. The $ float stops when either the first nonzero digit or the . or V is encountered. Again, the only exception is when the $ is written in all positions and the value is zero, in which case the entire field (including any . and ,) is blanked. If a comma happens to precede the first nonzero item, the comma is replaced by the dollar sign, which is the format generally desired for purposes of output.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 AMT PICTURE $9,999.99	2,350.22	$2,350.22
02 AMT PICTURE $9,999.99	150.31	$0,150.31
02 AMT PICTURE $$,999.99	150.31	bb$150.31
02 AMT PICTURE $$,$$$.99	25.40	bbb$25.40
02 AMT PICTURE $$,$$$.999	0.019	bbbbb$.019
02 AMT PICTURE $$,$$$.$$$	0.009	bbbbb$.009
02 AMT PICTURE $$,$$$.$$$	0.0	bbbbbbbbbb
02 AMT PICTURE $$,$$9.999	2,210.2	$2,210.200
02 AMT PICTURE $$,999.9	2,210.2	$2,210.2
02 AMT PICTURE $$,999.9	2,210.256	$2,210.2
02 AMT PICTURE $9,999.9999	23	$0,023.0000
02 AMT PICTURE $$,$$$.$$9	0.002	bbbbb$.002

The Z PICTURE Character

The Z PICTURE character is used to replace leading zeros by blanks and thus performs a function identical to that of the floating $ character, except for

insertion of the $ itself. As for the floating $, zero-suppression terminates when the first nonzero digit or the . character is encountered, whichever occurs first. As with the $ PICTURE character, the only exception occurs when Z's have been designated for all positions in a field and the value to be inserted in that field is zero, in which case the entire field is blanked. The following examples illustrate the use of the Z PICTURE character:

DESCRIPTION		NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 AMT PICTURE	Z99	25	b25
02 AMT PICTURE	ZZZ.99	25	b25.00
02 AMT PICTURE	ZZZ.99	0.10	bbb.10
02 AMT PICTURE	ZZZ.ZZZ	0.052	bbb.052
02 AMT PICTURE	ZZZ.ZZZ	0.0	bbbbbbb
02 AMT PICTURE	$ZZZ.9	13.2	$b13.2
02 AMT PICTURE	$ZZZZ.Z	13.2	$bb13.2
02 AMT PICTURE	$Z,ZZZ,ZZZ.ZZ	156,320.18	$bb156,320.18
02 AMT PICTURE	$Z,ZZZ,ZZZ.ZZ	3,156,320.18	$3,156,320.18
02 AMT PICTURE	$$,$$Z.ZZZ	0.001	bbbb$b.001

The + and − PICTURE Insertion Characters

Each of these editing characters can be inserted in the leftmost or rightmost position in a PICTURE. When the + character is used, any value that does not have an arithmetic sign associated with it is assigned a + sign. On the other hand, when the − PICTURE character is used, any value that does not have an arithmetic sign associated with it is represented in storage without a sign. In either case, a negative sign associated with a value always is represented in storage. The − PICTURE insertion character differs from the S character in that the use of the S character identifies a field as a signed one for computational purposes, but the sign does not occupy a position as such. Use of the − PICTURE character leads to a field in which the sign occupies a character position. The + character and the − character also can be floated; in this respect they are similar to the $ PICTURE character. However, the +, −, and $ are mutually exclusive as floating characters. If we want to have both $ float and + or − sign representation, we write the + or − to the right of the field, as illustrated in the last two of the following examples:

DESCRIPTION		NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 BALANCE PICTURE	+999.9	35.2	+035.2
02 BALANCE PICTURE	999.9+	35.2	035.2+
02 BALANCE PICTURE	999.9+	−35.2	035.2−
02 BALANCE PICTURE	++9.9	−001.3	b−1.3
02 BALANCE PICTURE	+++9.99	.05	bb+0.05
02 BALANCE PICTURE	+++9.99	−.05	bb−0.05
02 BALANCE PICTURE	++++.++	.01	bbb+.01
02 BALANCE PICTURE	−−−−−−−	0.0	bbbbbbb
02 BALANCE PICTURE	−−99.99	−10.25	b−10.25
02 BALANCE PICTURE	−999.99	100.25	b100.25
02 BALANCE PICTURE	999.9−	−10.2	010.2−
02 BALANCE PICTURE	$$$$.99−	20.35	b$20.35b
02 BALANCE PICTURE	$$$$.99+	20.35	b$20.35+

Review

1 Several $ signs included in a PICTURE clause signify that [several dollar signs should appear in the output / the output should contain the dollar sign in one of several possible positions].

the output should contain the dollar sign in one of several possible positions

2 Both the V PICTURE character and the . PICTURE character indicate __________ positions.

decimal point

3 The difference in the use of the V and the . PICTURE characters is that the V signifies an __________ decimal point, whereas the . signifies an __________ decimal point.

assumed; actual

4 In general, the $ float stops when the first nonzero digit is encountered or when the __________ PICTURE character is encountered. The only exception occurs when the value in the field is zero and the $ is written in all positions, in which case the field is filled with __________.

V *or* . *(decimal); blanks*

5 The PICTURE insertion character that is similar to the . PICTURE character but can appear more than once in a field is the __________ PICTURE character.

, *(comma)*

6 The character in a PICTURE clause which is used to replace with blanks the leading zeros in a value is the __________ PICTURE character.

Z

7 Representation of the algebraic sign of a numeric value is accomplished by the use of the __________ or __________ PICTURE character.

+; −

8 If the − PICTURE insertion character is used, a value held in storage will have associated with it either a − sign or [+ / no] sign. If the + PICTURE character is used, a value held in storage will have associated with it either a + sign or [− / no] sign.

no; −

The DB and CR PICTURE Characters

In accounting applications there is often need to identify values that represent debits or credits. The COBOL language facilitates such differentiation by means of the DB (debit) and CR (credit) editing characters. As indicated in the following examples, the DB or CR symbol is written only to the right of a field in the PICTURE clause, and in both cases it is represented in storage for the purpose of subsequent output only when the value is negative.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 RECEIPT PICTURE $999.99DB	135.26	$135.26bb
02 RECEIPT PICTURE $999.99DB	−135.26	$135.26DB
02 RECEIPT PICTURE $,$$9.99CR	−10.50	bb$10.50CR

Notice that the edited field does not provide for negative values as such. A value such as 10.50 would have been stored previously in a signed numeric field and then sent to the edited field. For example, if the original field is described by 03 PAY PICTURE S9(4)V99, then executing the instruction MOVE PAY TO RECEIPT will generate the stored content represented on the last line of the preceding table.

The following table summarizes the effects of the storage location associated with the use of the +, −, CR, and DB PICTURE editing symbols. Note that for positive values the + is included in the edited field only when the + PICTURE character appears in the PICTURE clause. Note also that for negative values the − sign is included in the edited field if either the + or − PICTURE character has been used, and that the CR or DB appears only if the numeric value is negative.

PICTURE CHARACTER USED	STORAGE REPRESENTATION WHEN VALUE IS POSITIVE	STORAGE REPRESENTATION WHEN VALUE IS NEGATIVE
+	+	−
−	Blank	−
DB	Blank	DB
CR	Blank	CR

The B PICTURE Character

This is an insertion editing character resulting in blanks being entered in the designated positions. For example, suppose the first two characters in the storage location NAME always represent the initials of a person's first name and middle name, as follows: RBSMITH. If we wish to print the name with spaces included between the two initials and between the initials and the last name, we can set up the editing field 02 EDNAME PICTURE ABABA(10). If we then execute the instruction MOVE NAME TO EDNAME and subsequently print the contents of EDNAME, the output will be R B SMITH.

The 0 PICTURE Character

The zero insertion character causes zeros to be inserted in the positions in which it appears. For example, we can use this option if the value represented in the storage is understood to be in thousands and we want to edit it to show the full value. Thus if we had 1365 as the value of sum and we set up EDSUM PICTURE 9(4)000, we could execute MOVE AMT TO EDSUM, giving the following result in EDSUM: 1365000.

The * PICTURE Character

The * character is referred to as a check-protect character and normally is used to protect dollar amounts written on checks or other negotiable documents. As indicated by the following examples, it works very much like the floating $ or the Z PICTURE character. In this case, however, instead of the $ sign being floated or positions being filled with blanks, the * character is entered in each zero-suppressed position as designated in the PICTURE clause.

DESCRIPTION	NUMERIC VALUE	REPRESENTED IN STORAGE AS
02 CHECK-VALUE PICTURE $***.99	256.18	$256.18
02 CHECK-VALUE PICTURE $***.99	10.13	$*10.13
02 CHECK-VALUE PICTURE $***.99	0.15	$***.15

The / PICTURE Character

Each / (stroke) in the PICTURE character string represents a character position into which the stroke character will be inserted. For example, suppose we have:

```
02 NUMERIC-DATE    PIC 9(6) VALUE 040786.
02 EDITED-DATE     PIC 99/99/99.
```

The instruction MOVE NUMERIC-DATE TO EDITED-DATE will cause EDITED-DATE to contain 04/07/86.

Review

1 The editing characters that can be used in a PICTURE clause to identify debits and credits, respectively, are the ________ and the ________ characters.

DB; CR

2 In order for a DB or CR to be included in an editing field, the value entered in that field must be [positive / negative / positive for DB but negative for CR].

negative

3 The insertion editing character that results in blanks being entered in the designated positions is the ________ PICTURE character, whereas the insertion editing character that results in zeros being entered in designated positions is the ________ PICTURE character.

B; 0

4 The insertion editing character that is referred to as the check-protect character is the ________ PICTURE character.

***

5 Use of the / (stroke) insertion character results in the stroke character being inserted in designated character positions [only when those positions are blank / to achieve visual separation of values].

to achieve visual separation of values

Summary of PICTURE Clause Options

Table 4-1 lists all the PICTURE characters. As indicated, the characters that identify the type of content in a storage field are the 9, A, and X characters. Special purpose characters associated with numeric fields only are the V, P, and S characters. All the other characters listed in Table 4-1 are used for editing purposes.

Instead of listing the characters that can be used in PICTURE clauses, another way of summarizing the material presented in this section is to consider the categories of data that can be contained in a storage location and the PICTURE characters that can be used with each category. Accordingly, Table 4-2 identifies five categories of data: numeric, alphabetic, alphanumeric, numeric edited, and

TABLE 4-1
TYPES OF CHARACTERS AVAILABLE FOR USE IN PICTURE CLAUSES

TYPE OF CHARACTER	SYMBOL	USE
Field definition characters	9	Numeric field
	A	Alphabetic field
	X	Alphanumeric field
Numeric field special character	V	Assumed decimal point
	P	Decimal scaling
	S	Operational (arithmetic) sign included
Editing characters	$	Dollar sign
	Z	Zero suppression
	*	Check protection
	.	Decimal point
	,	Comma
	+	Plus sign
	–	Minus sign
	DB	Debit
	CR	Credit
	B	Blank insertion
	0	Zero insertion
	/	Stroke insertion

TABLE 4-2
THE FIVE CATEGORIES OF DATA

Numeric items	The PICTURE may contain suitable combinations of the following characters: 9 V P and S.
Alphabetic items	The PICTURE clause contains only the A character.
Alphanumeric items	The PICTURE clause consists of A 9 and X characters. It cannot contain all A or all 9 characters, but it may contain a mixture of A and 9 characters.
Numeric edited items	The PICTURE clause can contain suitable combinations of the following characters: B P V Z O 9 , . + – CR DB $ and /.
Alphanumeric edited items	The PICTURE clause can contain combinations of the following characters: Z X 9 B 0 and /.

alphanumeric edited. Notice that the PICTURE clause for alphanumeric items cannot contain all 9s or all A's; all 9s would be indicative of a numeric field, and all A's would indicate an alphabetic field. Note also that numeric edited items can include appropriate combinations of all 12 editing characters included in Table 4-1. On the other hand, alphanumeric edited items can include the B and 0 (zero insertion) and / (stroke) editing characters only.

Review

1 Three of the PICTURE characters discussed are used for the purpose of defining the type of content in a storage field, namely, the ________, ________, and ________ characters. On the other hand, the three special characters used in conjunction with computational numeric fields are the ________, ________, and ________ PICTURE characters. (Refer to Table 4-1 if you wish.)

9, A, X; V, P, S

2 The only editing PICTURE character used in conjunction with an alphabetic field is the ________________ character, whereas the three editing characters that can be used in conjunction with an alphanumeric field are the ________________, ________________, and ________________ PICTURE characters.

B; B, 0, /

THE BLANK WHEN ZERO CLAUSE

Use of this clause achieves the same result as Z PICTURE, but it is more general. Consider the statement 02 AMOUNT PIC ZZ9.99 BLANK WHEN ZERO. If AMOUNT contains a zero value, the field will be blanked (six blanks); otherwise the PICTURE string will provide the editing.

Review

1 In the event that an entire data field contains a zero value, the field can be output as all blanks by use of the ________________ clause.

BLANK WHEN ZERO

CONDITION-NAMES

Recall that figurative constants are words that signify constant values. For example, the figurative constants ZERO and SPACES mean a value of zero and blanks, respectively. In effect, the use of condition-names enables the programmer to define additional figurative constants for use in the COBOL program. The use of this option always is indicated by the special level 88 entry, whose format is:

```
88 data-name { VALUE IS   } literal-1   [THRU literal-2]
             { VALUES ARE }

        [literal-3   [THRU literal-4]   ] . . .
```

As an example of the use of condition-names, suppose that the personnel record used in a company contains, among other things, the number of years of education. The information is contained in a field called EDUCATION and is so coded that the number indicates the last school grade completed. Thus, a code number less than 12 indicates that the person did not complete high school, 12 indicates a high school graduate, 13–15 indicate some college education, 16 indicates a college graduate, and a number greater than 16 indicates some graduate or postgraduate work. If we wish to process for educational level using these categories, we could write such PROCEDURE DIVISION statements as: IF EDUCATION IS LESS THAN 12 . . . IF EDUCATION IS EQUAL TO 12 . . . etc. However, an alternative is to define condition-names in the DATA DIVISION, which will then stand for the indicated values. Thus, we can write:

```
01  PERSONNEL-DATA.
    02  ID-NUMBER . . .
    02  NAME . . .
    02  ADDRESS . . .
    02  EDUCATION PICTURE IS 99.
        88  LESS-THAN-H-S-GRAD          VALUES ARE 0 THRU 11.
        88  H-S-GRAD                    VALUE IS 12.
        88  SOME-COLLEGE                VALUES ARE 13 THRU 15.
```

```
88  COLLEGE-GRAD          VALUE IS 16.
88  POST-GRAD             VALUES ARE 17 THRU 20.
88  ERROR-CODE            VALUES ARE 21 THRU 99.
```

When the preceding condition-names are defined in the DATA DIVISION, they then can be used in PROCEDURE DIVISION statements. For example, the statement IF H-S-GRAD ADD 1 TO TOTAL1 is identical in result to the statement IF EDUCATION IS EQUAL TO 12 ADD 1 TO TOTAL1. Furthermore, the statement IF SOME-COLLEGE ADD 1 TO TOTAL2 is equivalent to a series of *nested* conditional statements (see Chapter 6). An example is:

```
IF EDUCATION IS < 16
        IF EDUCATION > 12
            ADD 1 TO TOTAL2
                .
                .
```

Thus, one advantage of using condition-names is that they allow the programmer to write complex tests in simple form in the PROCEDURE DIVISION. At the beginning of this section, we indicated that the condition-name is like a figurative constant. After reviewing the previous example, it should be clear that this comparison holds true, except that the condition-name refers to a specific data-name only (EDUCATION in this case).

The use of condition-names improves the readability of a computer program and also makes it easier to change programs. Statements such as IF EDUCATION < 12 and IF SEX-CODE = 1 require that we remember the arbitrary meaning of the numeric codes. Their equivalent condition-names, however, are self-documenting when reference is made to LESS-THAN-H-S-GRAD or MALE. In terms of program changes, suppose that we decide to change the code for "male" from 1 to 2. In the absence of having used a condition-name, we would have to search the entire PROCEDURE DIVISION to make sure that every instance of the use of this code is changed. If the condition-name has been used, all we have to do is to change the definition of the condition-name from VALUE IS 1 to VALUE IS 2 in the DATA DIVISION.

Review

1 An entry in which a condition-name is defined is always assigned the level number ______ (number).

88

2 Suppose that for the data-name MARITAL-STATUS the possible values are 1 = married, 2 = divorced, 3 = widowed, 4 = single, and all other values are errors. Write suitable 88 level entries to define condition-names for

a condition of being or having been married
b condition of being single
c condition of error code.

03 MARITAL-STATUS PICTURE 9.
88 ______
88 ______
88 ______

a IS-OR-WAS-MARRIED VALUES ARE 1 THRU 3.
b SINGLE VALUE IS 4.
c ERROR-CODE VALUES ARE ZERO 5 THRU 9.

THE VALUE CLAUSE

In addition to defining storage fields with respect to form by using the PICTURE clause, it often is desirable to assign initial values to WORKING-STORAGE fields. Such a value may remain unchanged throughout the program, as in the case of a tax rate, or it may change in the course of program execution. Such initial values are not assigned to FILE SECTION items, since such fields receive their data either from the external medium or from some other storage location as the result of program execution.

As you may recall from previous examples, an initial value is assigned by the use of the VALUE clause. The general form of this clause is:

```
         { numeric literal      }
VALUE IS { figurative constant  }
         { nonnumeric literal   }
```

The use of the VALUE clause is illustrated in the following examples. Notice that the order of the PICTURE and VALUE clauses is irrelevant, and that the use of the word IS is optional for each type of clause.

```
02  PAGE-TITLE VALUES IS 'SAMPLE PROGRAM'     PICTURE IS A(14).
02  TENTH-BAL      PICTURE IS 99 VALUE 10.
02  TAX-RATE VALUE IS 0.03       PICTURE V99.
02  FILLER VALUE IS SPACES       PICTURE X(20).
02  ACCUMULATOR        PICTURE IS 9(8)V9(4) VALUE IS ZEROES.
```

At this point it should be emphasized that the VALUE clause associated with 88 level condition-names, as described in the preceding section, is different from the VALUE clause used to assign initial values to storage positions.

Review

1 Initial values can be assigned to WORKING-STORAGE fields by use of the ______________ clause.

VALUE

2 The VALUE clause [always precedes / always follows / can either precede or follow] the PICTURE clause in a WORKING-STORAGE entry.

can either precede or follow

SAMPLE DATA DIVISION

We conclude our discussion with a sample DATA DIVISION that includes many of the features discussed in this chapter. The sample is based on the salary-report programming task discussed as an illustration of program design at the end of Chapter 3. The task consists of reading records in an employee file and producing a report such as that illustrated in Figure 3-7. The input record layout chart is described in Figure 4-2, while the printer spacing chart is presented in Figure 4-3. Finally, Figure 4-4 presents a DATA DIVISION written for the programming task.

```
DATA DIVISION.
*
 FILE SECTION.
*
 FD  EMPLOYEE-FILE
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 80 CHARACTERS
     DATA RECORD IS EMPLOYEE-RECORD.
 01  EMPLOYEE-RECORD.
     02 EMPLOYEE-NAME            PIC X(15).
     02 EMPLOYEE-SEX-CODE        PIC 9.
        88 MALE                 VALUE 1.
        88 FEMALE               VALUE 2.
        88 ERROR-SEX-CODE       VALUE ZERO 3 THRU 9.
     02 EMPLOYEE-SALARY          PIC 9(5)V99.
     02 FILLER                   PIC X(57).
*
 FD  REPORT-FILE
     LABEL RECORDS ARE OMITTED
     RECORD CONTAINS 132 CHARACTERS
     DATA RECORD IS PRINT-LINE.
 01  PRINT-LINE                PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-TEST          PIC XXX VALUE 'NO'.
     88 END-OF-EMPLOYEE-FILE        VALUE 'YES'.
*
 01  EMPLOYEE-COUNTERS.
     02 NO-OF-MEN                PIC 99 VALUE ZERO.
     02 NO-OF-WOMEN              PIC 99 VALUE ZERO.
*
 01  SALARY-TOTALS.
     02 MEN-TOTAL-SALARY         PIC 9(7)V99 VALUE ZERO.
     02 WOMEN-TOTAL-SALARY       PIC 9(7)V99 VALUE ZERO.
 01  REPORT-HEADINGS.
     02 HEADING-1.
        03 FILLER                PIC X(36) VALUE SPACES.
        03 FILLER                PIC X(13) VALUE 'ANNUAL SALARY'.
*
     02 HEADING-2.
        03 FILLER                PIC X(16) VALUE SPACES.
        03 FILLER                PIC X(13) VALUE 'EMPLOYEE NAME'.
        03 FILLER                PIC X(10) VALUE SPACES.
        03 FILLER                PIC X(3)   VALUE 'MEN'.
        03 FILLER                PIC X(13) VALUE SPACES.
        03 FILLER                PIC X(5)  VALUE 'WOMEN'.
*
 01  REPORT-DETAIL-LINE.
     02 FILLER                      PIC X(15) VALUE SPACES.
     02 REPORT-EMPL-NAME            PIC X(15).
     02 REPORT-ERROR-CODE           PIC X(6) VALUE SPACES.
     02 REPORT-MEN-SALARY           PIC ZZ,Z99.99 BLANK WHEN ZERO.
     02 FILLER                      PIC X(8) VALUE SPACES.
     02 REPORT-WOMEN-SALARY         PIC ZZ,Z99.99 BLANK WHEN ZERO.
*
 01  REPORT-TOTALS-LINE.
     02 FILLER                      PIC X(15) VALUE SPACES.
     02 FILLER                      PIC X(9) VALUE 'T O T A L'.
     02 FILLER                      PIC X(9) VALUE SPACES.
     02 REPORT-TOT-MEN-SALARY       PIC $$,$$$,$99.99.
     02 FILLER                      PIC X(5)  VALUE SPACES.
     02 REPORT-TOT-WOMEN-SALARY PIC $$,$$$,$99.99.

 01  REPORT-AVERAGES-LINE.
     02 FILLER                      PIC X(15) VALUE SPACES.
     02 FILLER                      PIC X(13) VALUE 'A V E R A G E'.
     02 FILLER                      PIC X(5)  VALUE SPACES.
     02 REPORT-AVG-MEN-SALARY       PIC $$,$$$,$99.99.
     02 FILLER                      PIC X(5)    VALUE SPACES.
     02 REPORT-AVG-WOMEN-SALARY PIC $$,$$$,$99.99.
```

FIGURE 4-4
DATA DIVISION FOR THE SALARY REPORT PROGRAM.

Any DATA DIVISION is based on the analyst's understanding of the programming task and the record layout chart. For convenience, we selectively outline the DATA DIVISION below, discussing some of the highlights.

FD EMPLOYEE-FILE	The input file-name.
01 EMPLOYEE-RECORD.	The name of the input file record.
02 EMPLOYEE-SEX-CODE 88 MALE 88 FEMALE	In the data field identifying the sex category of the employee. Notice use of the 88 level numbers to specify the condition-names MALE and FEMALE.
88 ERROR-SEX-CODE	Another condition-name, which identifies the condition of EMPLOYEE-SEX-CODE having any of the values of zero or 3 thru 9.
01 END-OF-FILE TEST. 88 END-OF-EMPLOYEE-FILE	A data-name used for purposes of testing whether the end-of-file record has been read. Again, a condition-name is used for convenience in testing whether or not the end-of-file condition is true.
01 EMPLOYEE-COUNTERS. 02 NO-OF-MEN 02 NO-OF-WOMEN	The EMPLOYEE-COUNTERS group item is used for documentation purposes. There will be no reference to EMPLOYEE-COUNTERS as such in the PROCEDURE DIVISION. The references will be to either NO-OF-MEN or NO-OF-WOMEN, as appropriate in each case. These counter fields were created in recognition of the need to report the average salary, the computation of which requires a count of the number of men and women, respectively.
01 SALARY-TOTALS. 02 MEN-TOTAL-SALARY 02 WOMEN-TOTAL-SALARY	These data-names were defined in recognition of the fact that the required report presents the total salary for each of the two categories of employees.
01 REPORT-HEADINGS. 02 HEADING-1 . . . 02 HEADING-2	The report heading consists of two title lines. For better documentation we coined a group item REPORT-HEADINGS to refer to both of them, and then delineated each heading under HEADING-1 and HEADING-2. Blank lines between these headings can be specified by PROCEDURE DIVISION instructions (. . . ADVANCING . . . LINES), so we do not bother to specify blank line headings in the DATA DIVISION.
01 REPORT-DETAIL-LINE.	This item represents the format for producing each of the detail lines in the report.
02 REPORT-ERROR-CODE	This field is normally blank. However, as illustrated in the sample report in Figure 3-7 of the preceding chapter, we want to print six asterisks in any field for which the employee is not classified as being either male or female.
02 REPORT-MEN-SALARY	Notice the PIC clause illustrating the use of zero suppression, comma and decimal point insertion, and use of BLANK WHEN ZERO. The thinking is that if an employee record is for a woman, then we want spaces to be in the REPORT-MEN-SALARY, and vice versa for men. Therefore we plan to move zeros to the field, which then are converted to spaces by the BLANK WHEN ZERO

clause. (MOVE SPACES TO REPORT-MEN-SALARY would be improper because COBOL does not allow moving inconsistent categories of data. In this example, the alphanumeric data, SPACES, cannot be moved to the numeric edited field, REPORT-MEN-SALARY.)

01 REPORT-TOTALS-LINE.
.
.
.

These are specifications for the last two lines in the report, which constitute the footing, and include the total and average salary for each category.

01 REPORT-AVERAGES-LINE.
.
.
.
02 REPORT-AVG-MEN-SALARY

Notice the use of floating dollar signs and the larger field size. Since this is a sum, it should be larger than the field from which we add the data being accumulated. The BLANK WHEN ZERO clause (or the equivalent of $$,$$$,$$$.$$) is not used because we wish to print the zero rather than have a space if the total is in fact zero dollars.

So that this programming example will be relatively complete, Figure 4-5 presents the PROCEDURE DIVISION associated with the DATA DIVISION specifications in Figure 4-4. Study the PROCEDURE DIVISION statements particularly to observe how the data-names defined in the DATA DIVISION are used to accomplish the salary-report objective of the programming task. Notice that the paragraph names include a numeric prefix that corresponds to the order of the paragraphs in the program. This feature provides an immediate reference to the location of the paragraphs and is particularly useful for large programs.

```
PROCEDURE DIVISION.
*
 10-PROGRAM-SUMMARY.
    OPEN INPUT   EMPLOYEE-FILE
         OUTPUT  REPORT-FILE
*
    PERFORM 20-PRINT-REPORT-HEADINGS
*
    PERFORM 30-READ-EMPLOYEE-RECORD
*
    PERFORM 40-PROCESS-EMPLOYEE-RECORD
            UNTIL END-OF-EMPLOYEE-FILE
*
    PERFORM 80-PRINT-REPORT-FOOTINGS
*
    CLOSE EMPLOYEE-FILE
          REPORT-FILE
*
    STOP RUN.
*
 20-PRINT-REPORT-HEADINGS.
    WRITE PRINT-LINE FROM HEADING-1  AFTER ADVANCING PAGE
    WRITE PRINT-LINE FROM HEADING-2 AFTER ADVANCING 2 LINES
    MOVE SPACES TO PRINT-LINE
    WRITE PRINT-LINE AFTER ADVANCING 2 LINES.
*
 30-READ-EMPLOYEE-RECORD.
    READ EMPLOYEE-FILE RECORD
       AT END MOVE 'YES' TO END-OF-FILE-TEST.
*
 40-PROCESS-EMPLOYEE-RECORD.
    IF MALE PERFORM 50-PROCESS-MEN.
*
    IF FEMALE PERFORM 60-PROCESS-WOMEN.
```

FIGURE 4-5
PROCEDURE DIVISION FOR THE SALARY REPORT PROGRAM.

```
      *
           IF ERROR-SEX-CODE PERFORM 70-PROCESS-ERROR-CODE.
      *
           MOVE EMPLOYEE-NAME TO REPORT-EMPL-NAME
           WRITE PRINT-LINE FROM REPORT-DETAIL-LINE
                 BEFORE ADVANCING 1 LINE
      *
           MOVE SPACES TO REPORT-DETAIL-LINE
      *
           PERFORM 30-READ-EMPLOYEE-RECORD.
      *
       50-PROCESS-MEN.
           MOVE EMPLOYEE-SALARY TO REPORT-MEN-SALARY
           MOVE ZERO TO REPORT-WOMEN-SALARY
      *
           ADD 1 TO NO-OF-MEN
           ADD EMPLOYEE-SALARY TO MEN-TOTAL-SALARY.
      *
       60-PROCESS-WOMEN.
           MOVE EMPLOYEE-SALARY TO REPORT-WOMEN-SALARY
           MOVE ZERO TO REPORT-MEN-SALARY
      *
           ADD 1 TO NO-OF-WOMEN
           ADD EMPLOYEE-SALARY TO WOMEN-TOTAL-SALARY.
      *
       70-PROCESS-ERROR-CODE.
           MOVE ALL '*' TO REPORT-ERROR-CODE.
      *
       80-PRINT-REPORT-FOOTINGS.
           MOVE MEN-TOTAL-SALARY TO REPORT-TOT-MEN-SALARY
           MOVE WOMEN-TOTAL-SALARY TO REPORT-TOT-WOMEN-SALARY
           WRITE PRINT-LINE FROM REPORT-TOTALS-LINE
                 AFTER ADVANCING 3 LINES
      *
           DIVIDE NO-OF-MEN INTO MEN-TOTAL-SALARY
                  GIVING REPORT-AVG-MEN-SALARY
           DIVIDE NO-OF-WOMEN INTO WOMEN-TOTAL-SALARY
                  GIVING REPORT-AVG-WOMEN-SALARY
      *
           WRITE PRINT-LINE FROM REPORT-AVERAGES-LINE
                 AFTER ADVANCING 2 LINES.
```

FIGURE 4-5 *(Continued)*
PROCEDURE DIVISION FOR THE SALARY REPORT PROGRAM.

EXERCISES

4.1 True or false: The PICTURE clause can be used only with elementary items. Indicate the reason for your answer.

4.2 Indicate the size of each of the following fields in terms of the number of bytes:

PICTURE	SIZE
99V99	
9(3).9	
S999V9	
ZZ,ZZZ	
+(3).99	
$***,**9.99	
VPP99	
ZZZ000	

4.3 Complete the DATA DIVISION description for the following WORKING-STORAGE record so that it corresponds to the following record layout. Use your own choice of data-names.

```
01 TOTALS-LINE.
   02 ____________________ PIC ________ VALUE ________
   02 ____________________ PIC ________ VALUE ________
   02 ____________________ PIC ________ VALUE ________
   02 ____________________ PIC
   02 ____________________ PIC ________ VALUE ________
   02 ____________________ PIC ________
```

1–16	17–25	26–35	36–45	46–50	51–60
Blank	TOTAL	Blank	Men's total salary, showing decimal point and 2 decimal places	Blank	Women's total salary, showing decimal point and 2 decimal places

4.4 Referring to the following schematic representation, write a DATA DIVISION record description using the following information:

DEPT	2 letters
NAME	5 digits
RATE	4 digits, 2 decimal places, used for arithmetic
SKILL	1 letter
REGULAR	7 digits, 2 decimal places, used for arithmetic
OVERTIME	6 digits, 2 decimal places, used for arithmetic
SS-TAX	5 digits, 2 decimal places, used for arithmetic

<table>
<tr><td colspan="7">PAY-RECORD</td></tr>
<tr><td colspan="2">EMPLOYEE</td><td rowspan="3">RATE</td><td rowspan="3">SKILL</td><td colspan="3">YEAR-TO-DATE</td></tr>
<tr><td rowspan="2">DEPT</td><td rowspan="2">NAME</td><td colspan="2">GROSS</td><td rowspan="2">SS-TAX</td></tr>
<tr><td>REGULAR</td><td>OVERTIME</td></tr>
</table>

4.5 Write DATA DIVISION entries for the WORKING-STORAGE record named SALES-DATA whose description is given in the following table. Data are moved from the items whose PICTURE description is shown.

SOURCE ITEM PICTURE	RECEIVING ITEM-NAME	PRINT POSITIONS	EDITING REQUIRED
99999	SALE-NUMBER	1–5	Suppress all leading zeros.
		6–7	Blank
(X)25	NAME	8–32	None
		33–34	Blank
S9999V99	DOLLARS	35–?	Insert comma, decimal point. Dollar sign immediately to the left of leftmost nonzero digit. Show negative sign if negative.
		2 positions	Blank
S9(3)V9(4)	PROFIT		Show decimal point. Suppress leading zeros. Show negative sign to left of leftmost nonzero digit.

4.6 The following are DATA DIVISION entries for fields that contain the data to be printed as the CHECK-REGISTER record. The output resulting from printing of the CHECK-REGISTER record should have approximately the following format (header titles are shown for clarity only):

VENDOR NAME	VENDOR NUMBER	CHECK NUMBER	DATE	DEBIT	DISCOUNT	CASH
ACME CORP.	1234	12345	01/03/86	$1,030.57	$20.13	$1,010.44

```
01  CHECK-NUMBER        PICTURE 9(5).
01  DEBIT               PICTURE 9(6)V99.
01  DISCOUNT            PICTURE 9(4)V99.
01  CASH                PICTURE 9(6)V99.
01  VENDOR-DATA.
    02  V-NAME          PICTURE X(15).
    02  V-NUMBER        PICTURE X(4).
01  DATE.
    02  MONTH           PICTURE 99.
    02  DAY             PICTURE 99.
    02  YEAR            PICTURE 99.
```

Write DATA DIVISION entries to form the CHECK-REGISTER record so that the output is printed approximately in the desired format. (Make sure the data is in the form MM/DD/YY.)

4.7 It is required that two lines having the following general format be printed:

```
SUMMARY STATISTICS
AVERAGE BAL. $XXX,XXX.XX   MAX$XXX,XXX.XX   MIN$XXX,XXX.XX
```

Assume the following DATA DIVISION entries:

```
FD  PRINT-FILE LABEL RECORDS OMITTED DATA
        RECORD IS PRINT-LINE
01  PRINT-LINE      PICTURE X(132).
WORKING-STORAGE SECTION.
01  MAX-BAL         PICTURE 9(6)V99.
01  MIN-BAL         PICTURE 9(6)V99.
01  AVER-BAL        PICTURE 9(6)V99.
```

Write WORKING-STORAGE SECTION entries to set up the required fields to print these two lines.

4.8 Suppose it has become necessary to change an existing COBOL program. The original version of the relevant DATA DIVISION entries is as follows:

```
02  FIELD-A
    03  FIELD-B
    03  FIELD-C
    03  FIELD-D
02  FIELD-E
```

In the revised version it is required that the fields be restructured so that (a) reference can be made to all the fields as one unit; (b) reference can be made to fields B and C as a unit; and (c) reference can be made to fields D and E as a unit. Show how this can be done.

4.9 A wholesale distributor employs a number of salespeople in the five designated territories of a state. A coding method is used to denote a territory and a salesperson. The first digit indicates the territory and the second digit the individual salesperson, as shown in the following:

TERRITORY	CODE
Southwest	11–16
Northwest	21–25
Central	31–38
Northeast	41–49
Southeast	51–54

a Write condition-name entries so that, given a value of SALES-CODE, we can determine the territory and can identify an error code. For example, codes 08 and 19 would be error codes.

b For certain purposes, we are interested in testing whether a SALES-CODE value refers to the central territory or to the other territories. Set up condition-names that will enable us to test for central and noncentral territories.

4.10 Assume that in all cases the following two instructions apply:

```
MOVE A TO B
MOVE B TO C
```

where C has been defined as:

```
01 C PIC X(11).
```

Show the resulting contents in C, assuming each cell stands for one character position.

CONTENT OF A	PICTURE OF B	RESULTING CONTENT IN C										
10.125	999V99											
10000.00	Z,ZZZ.ZZ											
900.15	$$,$$Z.99											
0.08	$$,$ZZ.ZZ											
50.50	$***.99DB											
−25.25	+,+ + +.99											
25.25	$$$$.99−											
WELCOME	XXXBXXX											

5 PROCEDURE DIVISION STATEMENTS

■ COBOL LANGUAGE FORMATS

Before turning our attention to PROCEDURE DIVISION statements, we need to describe the method by which programming options are presented in this book. COBOL is characterized by great flexibility in the form of options available to the programmer. In order to communicate these options we use a *metalanguage*, a language about a language. The form of presentation used here is not unique to this book but generally is followed in all books concerned with COBOL program statements. The method is used in order to describe how each type of statement should be structured, and to identify the options available to the programmer for each type of statement. In other words, the style of presentation is necessary because we wish to talk about types of statements in general, rather than about specific and particular program instructions. For this purpose, then, the following set of conventions is followed:

1 Words presented entirely in uppercase letters are always reserved COBOL words.
2 Uppercase words that are underlined are words that are required in the type of program statement being described. Uppercase words that are not underlined are optional and are used only to improve the readability of the program.
3 Lowercase words are used to indicate the points at which data-names or constants are to be supplied by the programmer. In addition to the words "data-name" and "literal," the term "identifier" is used to indicate a data-name, but it has a slightly broader meaning. It refers to either of the following cases: data-names that are unique in themselves, or data-names that are not unique in themselves but are made unique through *qualification*. Qualification is discussed in Chapter 8: "Additional DATA and PROCEDURE DIVISION statements." For now, you may safely assume the words "data-name" and

"identifier" to be equivalent. Other lowercase words used to indicate items to be inserted by the programmer are:

file-name
record-name
integer
formula
condition
statement
any imperative statement
any sentence

4 Items enclosed in braces { } indicate that *one* of the enclosed items *must* be used.

5 Items enclosed in brackets [ ] indicate that the items are optional, and one of them *may* be used at the option of the programmer.

6 An ellipsis (. . .) indicate that further information may be included in the program instruction, usually in the form of repeating the immediately preceding element any desired number of times.

As an example of the use of these conventions, consider the ADD statement. With the COBOL language format, a basic form of ADD is:

```
ADD {identifier-1}   [, identifier-2]   . . . TO identifier-m
    {literal-1   }   [, literal-2   ]
```

If we apply the rules just presented, the word ADD is a reserved COBOL word because it is in uppercase, and it is required because it is underscored. The word TO is governed by the same rules. The braces following ADD indicate that one of the two alternatives enclosed *must* be used. Thus, the required word ADD must be followed by either an identifier or a literal. Incidentally, it also is understood that, for this specific instruction, the identifier must be an elementary numeric (nonedited) field and the literal must be a numeric literal. The square brackets indicate that identifier-2 and literal-2 are both optional. In other words, the identifier or literal that immediately follows ADD may or may not be followed by a second identifier or literal. The commas also are optional; they may be included to improve readability, or they may be omitted. The ellipsis indicates that the preceding element (in square brackets) may be repeated as many times as desired. Finally, the identifier-m indicates that there *must* be an identifier following the word TO. Note that it is not enclosed in braces, because it is the only option; braces are used when we may choose among alternatives.

Utilizing this general format, we see that the following examples are legitimate ADD statements:

```
ADD AMOUNT TO TOTAL
ADD 100 TO TOTAL
ADD REGULAR OVERTIME TO GROSS
ADD 10 BONUS 100.25 TO GROSS
```

As a final point, in this book the program statements and associated discussion materials that refer specifically to the revised language are always printed in color. For example, in the following format the NOT AT END and END-READ items are enhancements available in the new version, but not in the 1974 standard.

```
READ file-name RECORD [INTO identifier]
    [AT END imperative-statement-1]
    [NOT AT END imperative-statement -2]
    [END-READ]
```

Review

1 In presenting COBOL statement instructions, words that are entirely in uppercase letters designate ________________ words.

reserved COBOL

2 When a reserved COBOL word is underlined in the format presentation, this indicates that the word [may / must] be used as part of a program instruction.

must

3 Items to be inserted in the program instruction, such as data-names, identifiers, constants, and expressions, are indicated by [lowercase / uppercase] words.

lowercase

4 When two or more items are enclosed within brackets [ ], this indicates that one of them [may / must] be included in the program instruction. When two or more items are enclosed within braces { }, this indicates that one of them [may / must] be included.

may; must

5 To indicate that further information, such as additional data-names, can be included in an instruction, a(n) ________________ is used.

ellipsis

6 In this text, COBOL features available only with the revised language, and not with the 1974 standard, are highlighted by being printed in __________.

color

ARITHMETIC VERBS

Figure 5-1 presents the standard formats for the four principal arithmetic verbs: ADD, SUBTRACT, MULTIPLY, and DIVIDE. As can be seen in this figure, a variety of statement formats can be written. Before we present examples of how to use these verbs, we will discuss two specialized clauses that can be used with arithmetic staements: the ROUNDED and ON SIZE ERROR clauses.

The ROUNDED Clause

A need frequently exists for rounding numeric values. For example, even though prices or rates of interest may be quoted to three or four decimal places, any billing must be rounded to two decimal places, since the smallest monetary unit is the cent. COBOL provides automatic rounding by use of the ROUNDED clause, which can be used with all arithmetic verbs.

Execution of the statement ADD A TO B ROUNDED will result in a rounded number in B. If B were specified as containing two decimal places in the DATA DIVISION description, rounding would be accomplished by adding 0.005 to the result of the addition and truncating the third place. Therefore, when the remainder that is to be dropped begins with a 5 or higher value, the number is rounded up; otherwise it is rounded down. If B were specified to contain one place to the right of the decimal, 0.05 is added to the result of the addition, and the second place is truncated.

The ON SIZE ERROR Clause

The case may arise in which an arithmetic result is larger than anticipated, in terms of the number of digit positions available. For example, a construction

```
    {identifier-1} [identifier-2]
ADD {            } [            ] . . . TO identifier-m [ROUNDED]
    {literal-1   } [literal-2   ]

[identifier-n [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]

    {identifier-1} {identifier-2} [identifier-3]
ADD {            } {            } [            ] . . .
    {literal-1   } {literal-2   } [literal-3   ]

GIVING identifier-m [ROUNDED] [identifier-n [ROUNDED] ] . . .
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-ADD]

         {identifier-1} [identifier-2]
SUBTRACT {            } [            ] . . . FROM identifier-m [ROUNDED]
         {literal-1   } [literal-2   ]

[identifier-n [ROUNDED] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-SUBTRACT]

         {identifier-1} [identifier-2]          {identifier-m}
SUBTRACT {            } [            ] . . . FROM {            }
         {literal-1   } [literal-2   ]          {literal-m   }

GIVING identifier-n [ROUNDED]   [identifier-o [ROUNDED] ] . . .
[ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-SUBTRACT]

         {identifier-1}
MULTIPLY {            } BY identifier-2   [ROUNDED]
         {literal-1   }

[identifier-3 [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]

         {identifier-1}    {identifier-2}
MULTIPLY {            } BY {            } GIVING identifier-3 [ROUNDED]
         {literal-1   }    {literal-2   }

[identifier-4 [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-MULTIPLY]

       {identifier-1}
DIVIDE {            } INTO identifier-2 [ROUNDED]
       {literal-1   }

[identifier-3 [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

FIGURE 5-1
STANDARD COBOL FORMATS FOR THE FOUR ARITHMETIC VERBS.

```
       ⎧identifier-1⎫      ⎧identifier-2⎫
DIVIDE ⎨            ⎬ INTO ⎨            ⎬ GIVING identifier-3 [ROUNDED]
       ⎩literal-1   ⎭      ⎩literal-2   ⎭

[identifier-4 [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]

       ⎧identifier-1⎫    ⎧identifier-2⎫
DIVIDE ⎨            ⎬ BY ⎨            ⎬ GIVING identifier-3 [ROUNDED]
       ⎩literal-1   ⎭    ⎩literal-2   ⎭

[identifier-4 [ROUNDED] ] . . . [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]

       ⎧identifier-1⎫      ⎧identifier-2⎫
DIVIDE ⎨            ⎬ INTO ⎨            ⎬ GIVING identifier-3 [ROUNDED]
       ⎩literal-1   ⎭      ⎩literal-2   ⎭

REMAINDER identifier-4 [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]

       ⎧identifier-1⎫    ⎧identifier-2⎫
DIVIDE ⎨            ⎬ BY ⎨            ⎬ GIVING identifier-3 [ROUNDED]
       ⎩literal-1   ⎭    ⎩literal-2   ⎭

REMAINDER identifier-4 [ON SIZE ERROR imperative-statement-1]
[NOT ON SIZE ERROR imperative-statement-2]
[END-DIVIDE]
```

FIGURE 5-1 *(Continued)*
STANDARD COBOL FORMATS FOR THE FOUR ARITHMETIC VERBS.

worker earning $10.00 per hour should have a weekly gross pay well under $999.99. But suppose that by some mistake in the program, or more likely in the input, the computed weekly pay figure is over $1,000.00. Rather than allow truncation of this figure to occur, such ''overflows'' can be detected by use of the ON SIZE ERROR clause. For example, assume GROSS has PICTURE 999V99. We can write:

```
MULTIPLY RATE BY HOURS GIVING GROSS
    ON SIZE ERROR
    MOVE "GROSS PAY EXCEEDS $999.99" TO ERR-MESSAGE.
```

The ON SIZE ERROR clause is simply a conditional statement that says, if the size of a value does not fit in the field, do whatever is indicated in the statement that follows in that sentence. The statement that follows must be imperative; that is, it cannot be conditional. When ON SIZE ERROR is used and the condition is fulfilled, the arithmetic operand intended to receive the result is not altered from its previous value. In other words, it is as if the arithmetic operations had not happened.

In addition to ''large'' results, the ON SIZE ERROR condition also is fulfilled by a zero division. As you may recall from algebra, division by zero is an undefined operation yielding an ''infinitely'' large quotient.

In the revised language there are two enhancements that have been made with respect to the formats for each of the four arithmetic verbs presented in Figure

5-1. They are the NOT ON SIZE ERROR clause and the END-verb terminators. We will use an example as a means of presenting the reason for, and the effect of, the enhancements. Figure 5-2 presents code and corresponding flowcharts for purposes of contrasting the two versions. In the COBOL 1974 column notice that there is no period after MOVE ZERO TO Z, and as a result, the MOVE W TO Z is also executed when the ON SIZE ERROR is true. Let us recognize at this point that this is not the result intended; it illustrates programming error. The correct program statements in the 1974 version of the language are presented later in this section. The ON SIZE ERROR and other conditionals such as the AT END are actually specialized IF statements; in this vein, they apply until a period is encountered. In the 1974 version the ON SIZE ERROR provides for a course of action when the condition is true but it does not provide for a course of action when it is false.

The revised language allows for a symmetrical specification with the ON SIZE ERROR and NOT ON SIZE ERROR dual alternatives, as illustrated in the flowchart in the second column of Figure 5-2. The END-verb clause provides for termination of the SIZE ERROR conditional statement. Notice that in the example in Figure 5-2, there is no period after the END-ADD, and none is required. As the flowchart illustrates, the MOVE Z TO A is executed regardless of the truth or falsity of the SIZE ERROR condition.

COBOL 1974	REVISED VERSION
ADD X Y GIVING W ON SIZE ERROR MOVE ZERO TO Z MOVE W TO Z. MOVE Z TO A.	ADD X Y GIVING W ON SIZE ERROR MOVE ZERO TO Z NOT ON SIZE ERROR MOVE W TO Z END-ADD MOVE Z TO A.

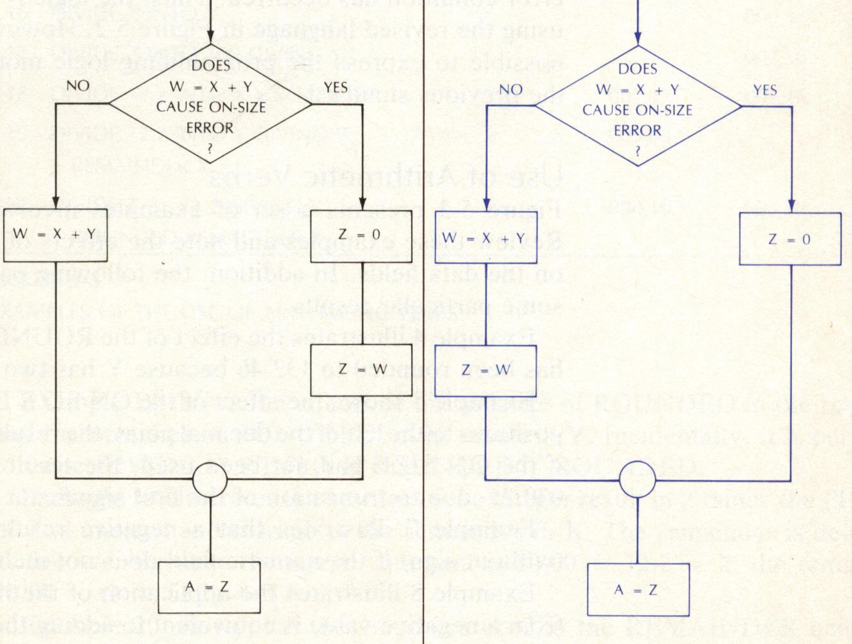


FIGURE 5-2
ON SIZE ERROR ILLUSTRATION.

As a final point, let us test our understanding of the difference in the two sample program codes in Figure 5-2. Suppose that in each case the following program statement preceded the code:

```
        .
     W  PIC 9.
        .
        .
     MOVE  3 TO W
     MOVE 10 TO X
     MOVE 20 TO Y
```

Deliberately, W has been defined with PIC 9 in the DATA DIVISION, which then invariably will result in the SIZE ERROR condition being true in the example above. What is the value stored in A at the completion of each program version in Figure 5-2? In the 1974 version, A is equal to 3, whereas in the revised version, A is equal to zero.

In case the above illustration suggests that COBOL 1974 has flaws, we again emphasize that the incorrect programming is not the fault of the language as such. Assuming that the programmer intended to apply the logic illustrated by the flowchart in the revised version column of Figure 5-2, then the program statements in the first column are incorrect. A correct sequence of statements to perform the task using the 1974 version of the language is:

```
MOVE 'NO' TO SIZE-ERROR-TEST
ADD   X   Y   GIVING   W
      ON SIZE ERROR MOVE 'YES' TO SIZE-ERROR-TEST.
IF SIZE-ERROR-TEST = 'YES'
      MOVE ZERO TO Z
ELSE
      MOVE W TO Z.
MOVE Z TO A.
```

In the illustration above we use a "Yes/No" test field to ascertain whether a size error condition has occurred. Thus, the logic is identical to the program segment using the revised language in Figure 5-2. However the revised language makes it possible to express the programming logic more directly and simply than does the previous standard.

Use of Arithmetic Verbs

Figure 5-3 presents a set of examples involving the use of arithmetic verbs. Review these examples and note the effects of the various forms of instructions on the data fields. In addition, the following paragraphs direct your attention to some particular results.

Example 4 illustrates the effect of the ROUNDED clause. The result of 132.456 has been rounded to 132.46 because Y has two decimal positions.

Example 5 shows the effect of the ON SIZE ERROR clause. Since Y has three positions to the left of the decimal point, the result 1030.25 is too large. Incidentally, if the ON SIZE had not been used, the result in Y would have been stored as 030.25, due to truncation of the first significant digit.

Example 7 illustrates that a negative result is stored as an absolute value (without sign) if the numeric field does not include the S PICTURE character.

Example 8 illustrates the application of the rule of arithmetic that subtracting from a negative value is equivalent to adding the number to be subtracted to that negative value.

Example 12 illustrates that truncation will occur if the number is larger than the defined field.

	W	X	Y	Z
PICTURE:	S999V99	999V99	999V99	999
INITIAL VALUE:	010‸00̄	090‸00	030‸00	040
1 ADD X TO Y			120‸00	
2 ADD X, Y TO Z				160
3 ADD 5 Y GIVING W	035‸00			
4 ADD X, 12.456 TO Y ROUNDED			132‸46	
5 ADD 1000.25 TO Y ROUNDED ON SIZE ERROR MOVE ZERO TO Z				000
6 SUBTRACT Y FROM X		060‸00		
7 SUBTRACT X FROM Y			060‸00	
8 SUBTRACT X Y FROM W	130‸00̄			
9 SUBTRACT W FROM X GIVING Y			100‸00	
10 SUBTRACT 1260.256 FROM Y ROUNDED ON SIZE ERROR MOVE ZERO TO Z				000
11 MOVE 10 TO X		010‸00		
MULTIPLY X BY Y			300‸00	
12 MULTIPLY X BY Y			700‸00	
13 MULTIPLY X BY Y ON SIZE ERROR MOVE ZERO TO Z				000
14 MULTIPLY Y BY 0.2453 GIVING X ROUNDED	007‸36			
15 DIVIDE Y INTO X		003‸00		
16 DIVIDE X INTO Y			000‸33	
17 DIVIDE Z INTO 100 GIVING Y			002‸50	
18 DIVIDE 12.2 INTO X Y ROUNDED		007‸37	002‸46	
19 DIVIDE 12.2 INTO Y GIVING Z REMAINDER X		005‸60		002
20 DIVIDE Z BY 12.2 GIVING Y ROUNDED REMAINDER X		000‸10	003‸28	

FIGURE 5-3
EXAMPLES OF THE USE OF ARITHMETIC VERBS.

Example 18 shows the effects of the absence of ROUNDED in the result in X and the presence of ROUNDED in the result in Y. Incidentally, it is permissible to write DIVIDE 12.2 INTO X ROUNDED Y ROUNDED.

Example 19 illustrates the storage of the integer result in Z (since the PICTURE of Z is integer) and storage of the remainder in X. The remainder is determined as follows. Because the *integer* quotient is 30.00 ÷ 12.2 = 2, the remainder is 30.00 − (2 × 12.2) = 5.6.

Example 20 shows that the value is stored in the REMAINDER field *before* the rounding takes place. Thus, the unrounded quotient is 3.278, which if stored in Y would have been stored unrounded as 3.27. The remainder is, then, 40 − (3.27 × 12.2) = 0.106, which is stored (right-truncated) in X as 0.10.

Review

1 If the result of an arithmetic operation is 45.4545, rounding to three decimal places will result in the value ________________ being placed in a storage location, whereas rounding to two places will result in the value ________ being placed in the storage location.

45.455; 45.45

2 If the ROUNDED option is not used, 45.4545 reported to three places would result in the value ________________, whereas the value reported to two places would be ____________.

45.454; 45.45

3 When the number of digits of an arithmetic result is greater than the number defined in the DATA DIVISION specifications, the likely reason is an error in the data input. Such an overflow can be signaled by use of the ________ clause.

ON SIZE ERROR

4 In addition to the availability of the ON SIZE ERROR clause, the revised language provides for a course of action when there is no such error by the ________________ clause.

NOT ON SIZE ERROR

5 In the revised langauge, termination of the SIZE ERROR statement, or processing associated with any verb, is signalled clearly by the ________ -verb clause.

END

6 Complete the following table by entering the numeric result of each arithmetic operation:

	W	X	Y	Z
PICTURE:	99V9	99	99V9	S999.9
INITIAL VALUE:	15‸0	10	12‸8	100‸0

1 ADD W, Y TO X
2 ADD W, Y GIVING X ROUNDED
3 SUBTRACT W FROM Y
4 SUBTRACT W FROM Y GIVING Z
5 MULTIPLY W BY Y
6 MULTIPLY X BY Y GIVING Z ROUNDED
7 DIVIDE X INTO Y ON SIZE ERROR MOVE ZERO TO X
8 DIVIDE X INTO Y ROUNDED REMAINDER Z

1 X = 37
2 X = 28
3 Y = 02‸2
4 Z = 002‸$\bar{2}$
5 Y = 92‸0
6 Z = 128‸8
7 Y = 1‸2
8 Y = 1‸3; Z = 008‸8 [= 12.8 − (1.2 × 10)]

THE COMPUTE VERB

Use of the four arithmetic verbs we have studied thus far is particularly suitable for single arithmetic operations, but suppose it is required that an answer be obtained by use of such a formula as $a = 3b - c + b(d - 2)$. If we were to use the four arithmetic verbs to solve this equation, a large number of statements would be required. However, use of the COMPUTE verb along with symbolic arithmetic operators makes it possible to write compact arithmetic statements for mathematical expressions.

Table 5-1 lists the symbols used for the arithmetic operations of addition, subtraction, multiplication, division, and exponentiation. Only the symbols for multiplication and exponentiation are different from the symbols commonly used in mathematics. In addition to the symbols, parentheses can be used to designate the order of operations; however, unlike their use in algebra, parentheses never are used to designate multiplication.

An arithmetic expression is formed by the use of arithmetic operators and data-names or literals. At least *one space must separate each operator symbol from the preceding and following data-names,* with parentheses used to designate or clarify the order of operations. Some examples of arithmetic expressions are:

ALGEBRAIC EXPRESSION	COBOL ARITHMETIC EXPRESSION
$a + b$	A + B
$a - b + (a - 5)c$	A - B + (A - 5) * C
$a^2 - \frac{b + c}{2}$	A**2 - (B + C) / 2

When parentheses are used, the operations within the parentheses are completed first, with order of priority given to the innermost sets, working from left to right in the arithmetic expression. In the absence of parentheses, the arithmetic operations are performed according to the following order of priority:

1. Exponentiation
2. Multiplication and division from left to right in the order written
3. Addition and subtraction from left to right in the order written

Consider the following COBOL examples:

COBOL ARITHMETIC EXPRESSION	ALGEBRAIC EXPRESSION
A + B / C	$a + \frac{b}{c}$
(A + B) / C	$\frac{a + b}{c}$
A + (B / C)	$a + \frac{b}{c}$

The first and third COBOL expressions represent the same algebraic expression. This is so because division takes priority over addition. Nevertheless, it is good

TABLE 5-1
THE FIVE ARITHMETIC OPERATIONS IN COBOL

+	Addition
−	Subtraction
*	Multiplication
/	Division
**	Exponentiation (raising to a power)

programming practice to include the parentheses in such cases, since documentation thereby is improved.

In addition to the five arithmetic operations, COBOL defines a *unary* operation. Standard COBOL defines both a "+" and a "−" unary operator. The operator is simply an instruction to multiply a variable by +1 or −1, respectively. Thus, if we want to multiply variable B times the negative value of variable A, we could use the unary operator as follows:

B * (−A)

The − is the unary operator. In this example, parentheses are used to avoid having two consecutive arithmetic operators.

Returning to the COMPUTE verb, we note that the general format associated with the use of this verb is:

```
COMPUTE {identifier-1 [ROUNDED]} . . . = arithmetic-expression-1
   [ON SIZE ERROR imperative-statement-1]
   [NOT ON SIZE ERROR imperative-statement-2]
   [END-COMPUTE]
```

A simple example of using the COMPUTE verb is:

```
COMPUTE GROSS = (REGULAR * WAGE) + 1.5 * (OVERTIME * WAGE).
```

An example that includes use of the ROUNDED and ON SIZE ERROR options is:

```
COMPUTE GROSS ROUNDED =
      (REGULAR * WAGE) + 1.5 * (OVERTIME * WAGE)
      ON SIZE ERROR
            PERFORM GROSS-TOO-BIG.
```

As indicated in the general format for the COMPUTE verb, and similar to the arithmetic verbs, two enhancements available in the revised language are the NOT ON SIZE ERROR clause and the END-COMPUTE terminator. These additional features are used for the same purposes as for the arithmetic verbs, as described in the preceding section of this chapter.

The identifier-1 in the COMPUTE format is the storage field that receives the results. It should be noted that it can be a numeric or numeric edited item. It really corresponds to the GIVING identifier clause in the other arithmetic verbs. All identifiers on the right-hand side, however, must be *elementary* numeric (nonedited) items.

The arithmetic operators +, −, *, and / correspond to the verbs ADD, SUBTRACT, MULTIPLY, and DIVIDE, respectively. The arithmetic operator ** has no corresponding verb and can be used only with the COMPUTE verb. Since exponentiation is a general mathematical process, it can be used to extract square roots as well as to raise numbers to various powers. Thus, A**2 means, a^2, but AA**0.5 means $\sqrt{A}$. This facility to extract roots increases the usefulness of the exponentiation operator. In general, however, COBOL has limited computational capabilities. COBOL programs are rarely written to accomplish tasks requiring complex computations. COBOL is used for data processing tasks rather than for computational tasks. Thus, logarithmic and trigonometric functions are not available in COBOL, although they commonly are available in other languages, such as FORTRAN and PL/1.

Review

1 As an alternative to the arithmetic verbs, arithmetic operators can be used in conjunction with the ________________ verb.

COMPUTE

2 The arithmetic symbols used with the COMPUTE verb which indicate the operations addition, subtraction, multiplication, division, and exponentiation are ________________, ________________, ________________, ________________, and ________________, respectively.

*+, −, *, /, and ***

3 The COBOL arithmetic expression corresponding to the algebraic expression $a^2 - 2ac + c^2$ is ________________.

(A**2) − (2 * A * C) + (C**2)
(See further comment in the next review item.)

4 Suppose that all the parentheses included in the above answer were omitted. The algebraic expression that corresponds to the resulting COBOL expression would be ________________.

$a^2 - 2ac + c^2$ *(Discussion continued in the next review item.)*

5 Therefore, because of the order in which the arithmetic operations always are performed, no parentheses in fact are required in the COBOL expression just given. However, such parentheses usually are included to improve readability of the program. In the absence of parentheses, the order of priority for the arithmetic operations is such that ________________ always is performed first, followed by ________________ and ________________, and culminating with ________________ and ________________.

exponentiation; multiplication; division; addition; subtraction

6 Typically, however, the use or nonuse of parentheses *does* make a difference in the way a COBOL arithmetic expression is evaluated. For each of the following COBOL expressions, indicate the equivalent algebraic expression:

COBOL ARITHMETIC EXPRESSION	ALGEBRAIC EXPRESSION
((A + (B * C)) / D)**2	
(A + (B * C)) / D**2	
A + (B * C) / D**2	
A + B * C / D**2	

$$\left(\frac{a+bc}{d}\right)^2$$

$$\frac{a+bc}{d^2}$$

$$a+\frac{bc}{d^2}$$

$$a+\frac{bc}{d^2}$$

7 An example of the use of the unary operator in a simple COBOL expression involving multiplication is ________________.

A * (− B)

8 In the general format associated with the COMPUTE verb, the results of the arithmetic operation are stored in [identifier-1 / arithmetic-expression-1].

identifier-1

FILE INPUT AND OUTPUT

The general functions performed by an input statement and an output statement are, respectively, to transfer data from an external storage medium to central storage, and to transfer data from central storage to an external storage medium. The file storage medium involved in each case is specified in the ENVIRONMENT DIVISION by the corresponding SELECT file-name and ASSIGN TO . . . statements.

The OPEN and CLOSE Verbs

Before a file can be used, we must OPEN that file and designate whether it is an input file or an output file. The format is:

```
OPEN { INPUT file-name-1    [file-name-2 . . . ] }
     { OUTPUT file-name-3   [file-name-4 . . . ] }
```

The meaning and purpose of the OPEN verb is explained in Chapter 9: "Sequential File Processing"; for now, we simply will say that it must be used prior to writing into or reading from a file.

We can use an OPEN statement for each file, or we can use one OPEN statement for several files as a group:

```
OPEN INPUT EMPL-FILE
OPEN OUTPUT REPORT-FILE
OPEN INPUT FILE-A FILE-B
     OUTPUT FILE-C FILE-D FILE-E.
```

Before a program terminates execution with a STOP RUN statement, we must use a CLOSE statement for each file that was opened. Again, the meaning and purpose of this statement will be explained further in Chapter 9.

The following is the format for the CLOSE verb; just as for the OPEN verb, we can use one CLOSE statement for each file or we can reference several files from one CLOSE.

```
CLOSE file-name-1    [file-name-2] . . .
```

The READ Verb

One format of the READ verb is:

```
READ file-name RECORD [INTO identifier]
[AT END imperative-statement-1]
[NOT AT END imperative-statement-2]
[END-READ]
```

Each execution of the READ transfers the data contained in the subsequent physical record of the file to the internal record described in the 01 entry that follows the FD specification. In effect, the READ is a "copy" command; it copies data from file storage to central storage.

COBOL 1974	Revised Version
MOVE 'NO' TO END-OF-FILE READ EMPLOYEE-FILE RECORD AT END MOVE 'YES' TO END-OF-FILE. IF END-OF-FILE = 'YES' PERFORM PRINT-SUMMARY ELSE PERFORM PRINT-DETAIL. PERFORM XYZ.	 READ EMPLOYEE-FILE RECORD AT END PERFORM PRINT-SUMMARY NOT AT END PERFORM PRINT-DETAIL END-READ PERFORM XYZ.

FIGURE 5-4
ILLUSTRATION OF READ VERSIONS.

The INTO option is a short form for moving the data from the file-record field to another field so that the data can be copied without a separate MOVE statement. Here are examples of two equivalent versions:

```
READ INPUT-FILE RECORD INTO OUTPUT-RECORD
    AT END MOVE 'YES' TO END-OF-FILE.
IF END-OF-FILE = 'NO'
    WRITE OUTPUT-RECORD . . .
```

The INTO OUTPUT-RECORD function could be replaced by the equivalent use of MOVE:

```
READ INPUT-FILE RECORD
    AT END MOVE 'YES' TO END-OF-FILE.
IF END OF FILE = 'NO'
    MOVE INPUT-RECORD TO OUTPUT-RECORD
    WRITE OUTPUT-RECORD . . .
```

Thus the INTO is simply a "short-hand" expression for MOVE . . .

The AT END is indicated as optional; however, until we discuss alternatives in Chapter 9, we will treat it as required. In each computer system there is a special character(s) for indicating the end of the file. When a record that contains such a character is read, the AT END condition is true and the imperative-statement is executed.

In the revised language the NOT AT END and the END-READ enhancements provide for an IF . . . ELSE . . . type of program structure, as illustrated in the two contrasting examples in Figure 5-4. In this figure, notice that there is no need to use a field such as END-OF-FILE in the revised version, since the AT END and NOT AT END allow for the dual alternatives to be expressed in the one READ statement.

The END-READ marks the termination of the READ statement. It could be omitted in the revised language provided that a period is placed after PERFORM PRINT-DETAIL. Such a period terminates the AT END conditional, and there is no ambiguity that PERFORM XYZ is to be executed regardless of whether AT END is true or false.

The WRITE Verb

The WRITE statement, as used with the printer, has the following format:

```
WRITE record-name          [FROM identifier-1]
 ┌                          { identifier-2 }        ┌LINE ┐ ┐
 │ { BEFORE }               { integer      }        └LINES┘ │
 │ {        } ADVANCING  {                                  }
 │ { AFTER  }               { mnemonic-name }               │
 └                          { PAGE          }               ┘
```

(In the format, WRITE, FROM, BEFORE, AFTER and PAGE are underlined.)

The FROM option is the counterpart of the INTO option for the READ verb. For example, we could write:

```
MOVE HEADER TO OUTPUT-RECORD
WRITE OUTPUT-RECORD . . .
```

or we could use the FROM:

```
WRITE OUTPUT-RECORD FROM HEADER . . .
```

The ADVANCING clause is used to control the vertical spacing on the printer. We already have seen examples of fixed spacing, such as when we write WRITE . . . AFTER ADVANCING 2 LINES. We also can have variable spacing, as the sample in the following paragraph illustrates.

Suppose that a sales invoice form has space for 10 items on each invoice, and we always wish to print the total billing on the last line of the invoice. We can use a storage field as a counter to keep track of the number of items on each invoice. Let us call this field KOUNT (note that we cannot use the data-name COUNT because this is a COBOL reserved word). If an invoice contains four items, the value stored in KOUNT will be 4; if an invoice contains six items, the value will be 6, and so on. In order to print the total billing on the last line of the invoice, the number of lines to be skipped will always be 10 spaces minus KOUNT. For this example, we of course are assuming that there will never be more than 10 items; therefore, the following statements provide the basis for skipping the appropriate number of lines:

```
SUBTRACT KOUNT FROM 10 GIVING LINE-COUNT
WRITE REP-LINE AFTER ADVANCING LINE-COUNT LINES.
```

When a mnemonic-name is used, it is specified in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. Such mnemonic names are specified by the manufacturer. As an example, consider a system where C12 is defined to mean channel 12 on the printer carriage control tape, where channel 12 defines the bottom of a printer page. We could have:

```
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER.
OBJECT-COMPUTER.
SPECIAL-NAMES.
        C12 IS BOTTOM-OF-PAGE.
INPUT-OUTPUT SECTION.
(etc.)
```

In the PROCEDURE DIVISION, the statement WRITE PRINT-RECORD AFTER ADVANCING BOTTOM-OF-PAGE is understood to mean to skip to the bottom of the page.

Review

1 Prior to inputting data from a file, a(n) ________________ statement must be executed.

OPEN INPUT

2 Prior to writing data from a file, a(n) ________________ statement must be executed.

OPEN OUTPUT

3 Before a program terminates execution with STOP RUN, a ____________ statement must be executed for every file that was opened.

CLOSE

4 Data are entered into central storage by use of the ____________ verb.

READ

5 The option that results in the same data being available in two different places after a READ statement has been executed (often in one input record and one output record) is the ____________ option.

INTO

6 Similar to the AT END option available with the READ statement, the revised language also makes available the ____________ option.

NOT AT END

7 The verb that is used to output data from central storage is the ____________ verb.

WRITE

8 In conjunction with a WRITE statement, data are moved from an identifier to an output record by use of the ____________ option.

FROM

9 Variable vertical spacing of output on the printer can be accomplished by using the ____________ clause in conjunction with a WRITE statement.

ADVANCING

ACCEPT AND DISPLAY VERBS

Up to this point, we always have discussed input and output in connection with files. Such is the normal use of input and output verbs, but it also is possible to execute input and output in conjunction with storage fields that are not part of any files. This typically is done to permit the input and/or output of short data items to or from devices such as the console typewriter and the printer. The verbs that allow such input and output are ACCEPT and DISPLAY. The general formats associated with use of the ACCEPT verb are:

Format 1

```
ACCEPT identifier [FROM mnemonic-name]
```

Format 2

```
                            ⎧ DATE        ⎫
                            ⎪ DAY         ⎪
ACCEPT identifier FROM      ⎨ DAY-OF-WEEK ⎬
                            ⎩ TIME        ⎭
```

The first format can be used as in this example: ACCEPT STARTING-CHECK-NO FROM CONSOLE-TYPE. As a result of executing this instruction, the computer will input data into the STARTING-CHECK-NO storage field from the device previously defined as CONSOLE-TYPE in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. If this device is a keyboard, further

program execution is delayed until the operator enters the appropriate input. If the device is a file, the next record will be read, and data will be input into STARTING-CHECK-NO.

Data input via ACCEPT is treated as if it were alphanumeric with respect to positioning. Suppose that STARTING-CHECK-NO has been defined with a PICTURE 99999. If ACCEPT is executed with reference to the console typewriter and the operator types in 12345, the receiving field will contain 12345. On the other hand, if the operator types 12, the 12 will be stored *left-justified* in STARTING-CHECK-NO, followed by three blanks. In general an entire record is moved from the input device to the identifier specified in ACCEPT. Thus when we typed 12 above, we were in effect transmitting a whole line from the console with blanks trailing the 12.

The second format of ACCEPT, given previously, can be used to move the contents of the COBOL predefined fields DATE, DAY, DAY-OF-WEEK, or TIME to a specified identifier. These latter fields are not defined by the programmer but are made available by the compiler. Their *implicit* definition is:

```
DATE          PIC 999999
DAY           PIC 99999
DAY-OF-WEEK   PIC 9
TIME          PIC 99999999
```

The DATE field contains the year, month, and day. Assume that TODAY was defined:

```
02  TODAY.
    03  T-YEAR    PIC 99.
    03  T-MONTH   PIC 99.
    03  T-DAY     PIC 99.
```

The instruction ACCEPT TODAY FROM DATE issued on February 1, 1986, will cause the content of DATE, 860201, to be moved to TODAY.

The DAY field contains the year and the day of the year, counting from January 1 as 001 to December 31 as 365 (non leap year). Thus, February 1, 1986, will be stored as 86032.

TIME contains hours, minutes, seconds, and hundredths of a second, based on elapsed time after midnight on a 24-hour-clock. Thus, 8:30 P.M. is stored as 20300000. The smallest value of TIME is 00000000, and the largest is 23595999.

The DAY-OF-WEEK yields a one-digit code that corresponds to the days of the week as follows:

1 = Monday, 2 = Tuesday, . . . 7 = Sunday.

If we define:

```
02 DAY-NAME-CODE PIC 9.
   88  MONDAY VALUE 1.
   88  TUESDAY VALUE 2.
   .
   .
   .
   88 SUNDAY VALUE 7.
```

Then we can use program statements like the following:

```
ACCEPT DAY-NAME-CODE FROM DAY-OF-WEEK
IF MONDAY PERFORM MONDAY-ROUTINE.
IF TUESDAY PERFORM TUESDAY-ROUTINE.
.
.
.
```

The general format associated with use of the DISPLAY verb is:

```
DISPLAY { identifier-1 } [ , identifier-2 ] . . . [UPON mnemonic-name]
        { literal-1    } [ , literal-2    ]
    [WITH NO ADVANCING]
```

Notice that DISPLAY can reference a series of identifiers or literals. Thus, we can write DISPLAY AMOUNT-A, 'IS A VALUE OUT OF RANGE' UPON CONSOLE-TYPE. Execution of this statement will result in the contents of the storage field AMOUNT-A being output on the device named CONSOLE-TYPE, followed by the literal message in the quotation marks.

In some computer installations, ACCEPT and DISPLAY can be used only with devices predefined by the installation itself; thus, the device will not be named in the statement. For example, writing ACCEPT CODE-A may be valid and would refer to some specific device in the particular installation, such as the console. Similarly DISPLAY ERR-MESSAGE-1 may be valid in installations in which the DISPLAY verb is associated automatically with a particular device, such as the console or a specific printer.

A common use of DISPLAY is for program debugging purposes, especially if one is in a system in which execution of the DISPLAY command results in the output being displayed at the video monitor or an easily accessible printer. For instance, if we encounter an error in results associated with a field MEN-TOTAL-SALARY, we may want to DISPLAY the contents of this field each time a value is added to it, as in the following example:

```
DISPLAY 'MEN-TOTAL-SALARY BEFORE ADDING = ' MEN-TOTAL-SALARY
ADD EMPLOYEE-SALARY TO MEN-TOTAL-SALARY
DISPLAY '                 AFTER ADDING = ' MEN-TOTAL-SALARY
```

These statements allow the programmer to trace the accumulation of values in the target field and, perhaps, provide a clue as to the cause of the error. We could, of course, use the WRITE verb to do the same thing, but it would require additional specifications and commands. A WORKING-STORAGE group item consisting of two elementary fields would have to be defined: one field for the explanatory header in the above examples and one for the content of MEN-TOTAL-SALARY. Then we would MOVE and WRITE the appropriate data.

In the revised language, a new enhancement of DISPLAY has been introduced that includes the WITH NO ADVANCING option. This option is useful when writing on a video monitor because it prevents the cursor from moving to the next line, which is the normal action after a DISPLAY command.

This is an appropriate point to discuss how writing on a CRT device is controlled. In particular, let us consider how the specific positions of input and output on the screen are designated and how special visual attributes, such as bold-face, blinking, and reversed video characters are achieved. The general procedure is to send special codes to the terminal that specify the desired action. The terminal then sorts out the special codes, does what these codes specify, and then displays the (remaining) data. It will be easiest to consider an example. By necessity the example is hypothetical because the specific procedure and codes depend on both the computer and the terminal.

Let us assume the following (fictitious) codes:

= the "escape" character. When the terminal receives such a character it is recognized as a special code that marks the beginning of a command string.

The escape character plus the one or more characters that follow are not displayed. They are understood to comprise a command to the terminal. The character immediately following the escape character determines the length of the command string.

@ = position the cursor to row/column position designated by the following two 2-digit numbers

? = turn on the normal display mode.

! = turn on the bold-face display mode.

Next we define the following fields:

```
02  CURSOR-POSITION.
    03  ESCAPE-CODE        PIC X          VALUE '#'.
    03  CURSOR-CODE        PIC X          VALUE '@'.
    03  ROW-POSITION       PIC 99.
    03  COLUMN-POSITION    PIC 99.
02  DISPLAY-FIELD.
    03  ESCAPE-CODE        PIC X          VALUE '#'.
    03  PRINT-MODE         PIC X.
    03  MESSAGE-FIELD      PIC X(40).
02  INPUT-FIELD   PIC   X(20).
```

In order to write the words "Type your NAME:" in row 10, column 20, we would write:

```
MOVE 10 TO ROW-POSITION
MOVE 20 TO COLUMN-POSITION
DISPLAY CURSOR-POSITION WITH NO ADVANCING
MOVE '?' TO PRINT-MODE
MOVE 'Type your NAME:' TO MESSAGE-FIELD
DISPLAY DISPLAY-FIELD.
```

In order to move the cursor to row 11, column 30, and to receive whatever data is entered in the keyboard we would write:

```
MOVE 11 TO ROW-POSITION
MOVE 30 TO COLUMN-POSITION
DISPLAY CURSOR-POSITION WITH NO ADVANCING
ACCEPT INPUT-FIELD.
```

Finally, to print the word 'BOLD' in column 1, row 1, in boldface mode and to leave the cursor on the same line, we would write:

```
MOVE 1 TO ROW-POSITION
MOVE 1 TO COLUMN-POSITION
DISPLAY CURSOR-POSITION WITH NO ADVANCING
MOVE '!' TO PRINT-MODE
MOVE 'BOLD' TO DISPLAY-FIELD
DISPLAY DISPLAY-FIELD WITH NO ADVANCING.
```

The above discussion and illustrations are designed to present the general concept. Each reader will have to use the specific methods and codes applicable to the equipment being used. Unfortunately, interactive screen handling is not a standardized COBOL feature.

Review

1 The verb used to input data items not usually part of any file as such is the ________________ verb, whereas the verb similarly used for output, often on a CRT terminal, is ________________.

ACCEPT- DISPLAY

2 DATA input by use of the ACCEPT verb is always treated as being [alphabetic / numeric / alphanumeric].

alphanumeric

3 Because the DISPLAY verb can reference literals as well as identifiers, it can be used to convey certain ______________ as well as data items.

messages

4 Although the DISPLAY verb has limited use in computer programs for regular data processing applications, it is used frequently in conjunction with the ______________ of computer programs.

debugging

5 Such special effects as boldface are obtained on a CRT terminal by sending special ______________ to the terminal.

codes

6 Each code that is to be used to achieve a special effect on a CRT terminal is defined in the DATA DIVISION, in the ______________ clause for the data-item designating that effect.

VALUE

DATA TRANSFER

Data transfer in COBOL is accomplished by use of the MOVE verb. Actually, both the term "data transfer" and the word MOVE are misnomers, because the function performed is *duplicating,* not moving. When we write MOVE A to B, the contents of A are duplicated in B, with the contents of A remaining unchanged.

The format for the MOVE instruction is:

```
       {identifier-1}
MOVE   {            }  TO identifier-2   [, identifier-3] . . .
       {literal     }
```

In any use of the MOVE instruction, caution should be exercised so that the receiving field is not smaller than the sending field. If the receiving field is too small, truncation will result in the loss of part of the value or information being transferred. The rules for moving data are many and relate to the form of data moved. One important point to remember is the following: numeric data are aligned according to the decimal point. If the receiving field is larger than the sending, the extra positions are filled with zeros. If the receiving field is smaller, then truncation takes place as needed, to the right, left, or both right and left of the decimal point. In the following illustration the caret (‸) implies a decimal point:

SENDING FIELD	RESULT IN RECEIVING FIELD
0 1 3‸5 2	0 0 1 3‸5 2 0
2 5 2 3‸5	2 5 2 3‸5
2 5 2 3‸5	5 2 3‸5
2 5 2 3 5‸	5 2 3‸

A second important point is that alphabetic or alphanumeric data are left-justified in the receiving field (unless the programmer uses the JUSTIFIED RIGHT clause in the DATA DIVISION). If the receiving field is larger than the sending field, the additional positions are filled with blanks. If the receiving field is smaller, then truncation takes place from the right. The following examples illustrate some typical cases:

SENDING FIELD	RECEIVING FIELD
B O N A N Z A	B O N A N Z A ␣ ␣
B O N A N Z A	B O N A N Z A
B O N A N Z A	B O N A N

Table 5-2 indicates the types of data transfer that are and are not permitted in COBOL. The reader may want to review Table 4-2 in the preceding chapter, which defines the various categories of data to which reference is made in Table 5-2. Most compilers have incorporated the rules presented in this table, so that the programmer will be warned of illegal MOVE statements; still, it is not difficult to internalize most of these rules.

Refer to the last line of Table 5-2. In the revised language it is permissible to move a numeric edited field to a numeric unedited field. This new option allows for *de-editing* of data, something not permitted in the 1974 version. As an example consider these statements:

```
02  COMP-AMOUNT    PIC  9(5)V99.
02  EDIT-AMOUNT    PIC  $$,$$9.99.
        .
        .
        .
MOVE  1234.56 TO EDIT-AMOUNT
MOVE  EDIT-AMOUNT TO COMP-AMOUNT.
```

When these statements have been executed, EDIT-AMOUNT would contain $1,234.56 while COMP-AMOUNT would correctly contain 0123456. The last statement, MOVE EDIT-AMOUNT to COMP-AMOUNT strips the data of such editing characters as the dollar sign, the comma, and the decimal point, and moves the numeric data to the numeric unedited field. Such a move is not valid in the 1974 version of COBOL.

TABLE 5-2
LEGAL AND ILLEGAL MOVE COMMANDS

	CATEGORY OF RECEIVING DATA ITEM		
CATEGORY OF SENDING DATA DATA ITEM	ALPHABETIC	ALPHANUMERIC; ALPHANUMERIC EDITED	NUMERIC INTEGER; NUMERIC NONINTEGER; NUMERIC EDITED
Alphabetic	Yes	Yes	No
Alphanumeric	Yes	Yes	No
Alphanumeric edited	Yes	Yes	No
Numeric integer	No	Yes	Yes
Numeric noninteger	No	No	Yes
Numeric edited	No	Yes	No Yes (in revised language)

It is often assumed that the DATA DIVISION description of a field as being numeric somehow guards against the moving of non-numeric data into that field. We present the following example to dispel such a notion.

```
01  SAMPLE-FIELD.
02  A  PIC  XX.
02  B  PIC  99V99.

MOVE 'ROBERT' TO SAMPLE-FIELD.
```

After the MOVE is executed, B contains the letters "BERT"! How can that be? It is very simple. SAMPLE-FIELD is a group-item which is six bytes long. Therefore, when we move 'ROBERT' into it, the last four bytes of the field contain the last four characters in the literal.

The above example constitutes a valid action in COBOL and shows the flexibility that the language provides. On the other hand, it would not be correct to proceed to do arithmetic involving the B field after the above MOVE. For example,

```
MOVE 'ROBERT' TO SAMPLE-FIELD
ADD 1 TO B
```

would be an incorrect program as far as logic is concerned. However from the standpoint of COBOL the ADD is a valid statement, because it is phrased independently of the content of B.

Many compilers add program code to check whether a field does contain numeric data before doing arithmetic with that field. The reader may wish to test the above illustration on an available compiler to see what happens.

To guard against such possible errors, COBOL provides for class condition tests, discussed in Chapter 6, "Conditions and Conditional Statements." The following illustrates such a class condition test:

```
IF B IS NUMERIC
    ADD 1 TO B
ELSE
    PERFORM ERROR-ROUTINE.
```

Review

1 When data are transferred by use of the MOVE instruction, one might say more correctly that the data have been ________, rather than "moved."

duplicated

2 When numeric data are moved from a sending field to a receiving field, alignment takes place with respect to the ________.

decimal point

3 When nonnumeric data are moved from a sending field to a receiving field, alignment takes place at the [left / right] margin.

left

4 Movement of numeric data into a numeric edited field is valid [in both versions of COBOL / only in the new revision].

in both versions of COBOL

5 Movement of numeric edited data into a numeric field is valid [in both versions of COBOL / only in the new revision].

only in the new revision

6 Non-numeric data can be moved inadvertently into a numeric field when the field is part of a ________________-item that is alphanumeric, for example.

group

THE PERFORM VERB

In general, the PERFORM verb is used to specify the sequence of execution of a modular program, represented by paragraphs or sections, or to specify the repeated execution of such modules to achieve program looping. We already have seen the use of the PERFORM verb in the program examples in the preceding chapters. In this section, we review and expand our description of the uses of this verb by presenting the basic formats associated with the PERFORM verb. This is a highly flexible verb, and additional formats for its use will be presented in later chapters.

The simplest format of the PERFORM is:

```
PERFORM    procedure-name
```

The procedure-name referenced in this format is either a paragraph- or section-name. Simply stated, the PERFORM directs program execution to the named procedure, at which point the procedure is executed and program execution returns to the next statement following the PERFORM. In this respect, it follows that the named procedure must allow the program to return; that is, the procedure must not contain a GO TO or a STOP RUN statement. However, the paragraph that is performed may itself contain another PERFORM instruction.

The PERFORM . . . UNTIL

Another format of PERFORM is:

```
PERFORM procedure-name

    [ WITH TEST { BEFORE } ]   UNTIL condition
                { AFTER  }

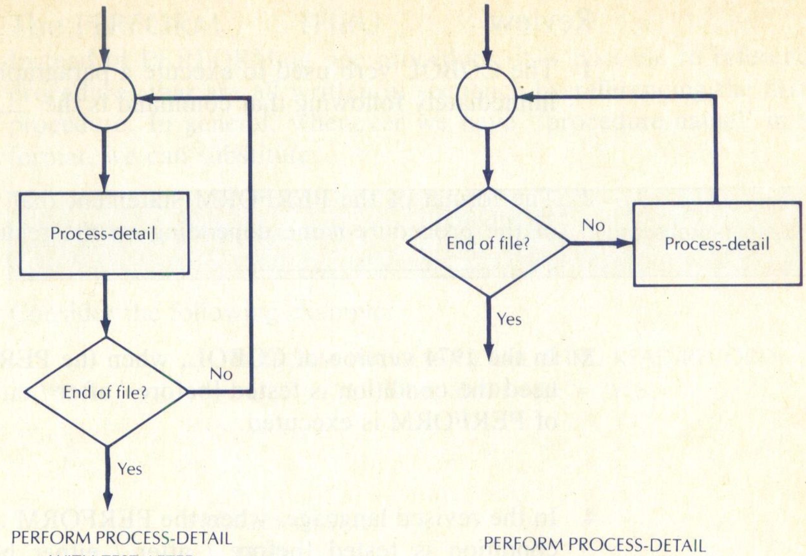
[END-PERFORM]
```

In its simplest form, this format is illustrated by:

```
PERFORM PROCESS-CUSTOMER-DETAIL
        UNTIL END-OF-FILE = 'YES'.
```

In the program segment above the UNTIL condition is tested *before* the procedure is executed. The END-OF-FILE first is tested for 'YES'. If END-OF-FILE is equal to YES, the program continues with the next statement. If it is *not* equal to YES then PROCESS-CUSTOMER-DETAIL is executed, and the process is repeated until END-OF-FILE = 'YES'.

The revised language allows for greater control over the repeated execution of a procedure. If we want to do a procedure and *then* determine if it should be done again, we use the WITH TEST AFTER option. If we want to determine if a procedure should be done beforehand, then we can either use the TEST BEFORE option or leave the TEST option out altogether. Figure 5-5 presents a flowchart and an example of each of the two TEST options in the PERFORM . . . UNTIL statement.




```
PERFORM PROCESS-DETAIL
    WITH TEST AFTER
    UNTIL END-OF-FILE = 'YES'
```

```
PERFORM PROCESS-DETAIL
    WITH TEST BEFORE
    UNTIL END-OF-FILE = 'YES'
```

FIGURE 5-5
PERFORM. . .UNTIL WITH USE OF TEST OPTION.

The In-Line PERFORM

It is also possible to use an *in-line* PERFORM, which does not make reference to a procedure-name. As an example, consider the case in accounting in which we need to form the "sum of the digits" for depreciating an asset over N years, where N has an integer value. We need to form the sum of 1 + 2 + 3 + . . . + (N − 1) + N. Thus, if N is 5, we have 1 + 2 + 3 + 4 + 5. Here is a routine for doing this task:

```
MOVE ZERO TO SUM-OF-DIGITS
MOVE 1 TO YEAR-COUNTER
PERFORM UNTIL YEAR-COUNTER > N
    ADD YEAR-COUNTER TO SUM-OF-DIGITS
    ADD 1 TO YEAR-COUNTER
END-PERFORM.
```

Notice the absence of a procedure-name between the words PERFORM and UNTIL. This signifies that the command(s) to be performed follows the basic PERFORM UNTIL statement and is marked by the END-PERFORM terminator. Thus, the two ADD statements constitute the procedure to be executed in the example above. This format of the PERFORM allows for a "nameless" procedure, in essence. Contrast the above example with the equivalent one below:

```
MOVE ZERO TO SUM-OF-DIGITS
MOVE 1 TO YEAR-COUNTER
PERFORM FORM-SUM UNTIL YEAR-COUNTER > N
        .
        .
        .
FORM-SUM.
  ADD YEAR-COUNTER TO SUM-OF-DIGITS
  ADD 1 TO YEAR-COUNTER.
```

We can see that the in-line PERFORM is more convenient when, as in this example, the procedure is so simple that it does not warrant setting up a separate paragraph.

Review

1 The COBOL verb used to execute a paragraph and return to the instruction immediately following that command is the ________________ verb.

PERFORM

2 The format of the PERFORM statement that permits *conditional* execution of the procedure-name depending on the result of a test is the PERFORM . . . ________________.

UNTIL

3 In the 1974 version of COBOL, when the PERFORM . . . UNTIL format is used the condition is tested [before / after / either before or after] the object of PERFORM is executed.

before

4 In the revised language, when the PERFORM . . . UNTIL format is used the condition is tested [before / after / either before or after] the object of PERFORM is executed.

either before or after

5 The version of the PERFORM command which does not make reference to a procedure-name, but rather the commands to be executed immediately following the PERFORM UNTIL conditional statement, is the ________ PERFORM.

in-line

The PERFORM . . . TIMES

There are tasks in which a procedure needs to be executed a definite number of times, and in such cases the TIMES option is useful.

```
PERFORM   [procedure-name]   {integer   }   TIMES
                             {identifier}

     [imperative-statement]
[END-PERFORM]
```

For the sum-of-digits example discussed above, we could write:

```
        MOVE ZERO TO SUM-OF-DIGITS
        MOVE 1 TO YEAR-COUNTER
        PERFORM FORM-SUM N TIMES
        .
        .
        .
    FORM-SUM.
        ADD YEAR-COUNTER TO SUM-OF-DIGITS
        ADD 1 TO YEAR-COUNTER.
```

Alternately, using the in-line PERFORM of the revised language, we could write:

```
MOVE ZERO TO SUM-OF-DIGITS
MOVE 1 TO YEAR-COUNTER
PERFORM N TIMES
    ADD YEAR-COUNTER TO SUM-OF-DIGITS
    ADD 1 TO YEAR-COUNTER
END-PERFORM.
```

The PERFORM . . . THRU

Instead of PERFORMing one procedure, it is possible to reference a *series* of procedures that are all written in sequence by referencing the first and the last procedure. In general, whenever we have "procedure-name" in a PERFORM format, we can substitute:

```
PERFORM  procedure-name-1  {THROUGH}  procedure-name-2
                           {THRU   }
```

Consider the following example:

```
PERFORM PROCESS-DETAIL THRU READ-RECORD
        UNTIL END-OF-FILE = 'YES'.
.
.
.
PROCESS-DETAIL.
    ADD . . . .
    MOVE . . . .
    WRITE . . . .
COMPUTE-SPECIAL-PRICE.
    IF . . .
    MOVE . . .
READ-RECORD.
    READ CUSTOMER-FILE
        AT END MOVE 'YES' TO END-OF-FILE.
```

The example in effect states that the group of paragraphs that begin with the one named PROCESS-DETAIL through the one named READ-RECORD are to be performed. When the last statement in the last-named paragraph is executed, then program execution returns to the original PERFORM statement. In everyday language, the PERFORM . . . THRU states:

> Perform the group of consecutive paragraphs that begins with the first-named paragraph and ends with the second-named paragraph.

Although the THRU option is available, we do not recommend its use except for very special circumstances. Although it is a convenient option, it is also an error-prone option. It relies heavily on the *physical* position of paragraphs. For instance, in the above example, if the COMPUTE-SPECIAL-PRICE paragraph were moved elsewhere in the program, a most serious error would result. As we consider the three paragraphs PROCESS-DETAIL, COMPUTE-SPECIAL-PRICE, and READ-RECORD, it is difficult to see any relationships among them. It is not clear that after the WRITE in PROCESS-DETAIL we would begin executing the IF in the COMPUTE-SPECIAL-PRICE paragraph.

Instead of using the THRU option, we recommend using one paragraph which in turn includes PERFORM commands that make reference to each of the other paragraphs. Using such an approach for the above example, we have:

```
PERFORM PROCESS-CUSTOMER
        UNTIL END-OF-FILE = 'YES'.
.
.
.
PROCESS-CUSTOMER.
    PERFORM PROCESS-DETAIL
    PERFORM COMPUTE-SPECIAL-PRICE
    PERFORM READ-RECORD.
```

```
PROCESS-DETAIL.
    ADD . . .
COMPUTE-SPECIAL-PRICE.
    IF . . .
READ-RECORD.
    READ CUSTOMER-FILE . . .
```

In this revised structure, the physical position of paragraphs is irrelevant, and the relationship among the paragraphs is clear.

Alternatively, we can use an in-line PERFORM, as follows:

```
PERFORM UNTIL END-OF-FILE = 'YES'
    PERFORM PROCESS-DETAIL
    PERFORM COMPUTE-SPECIAL-PRICE
    PERFORM READ-RECORD
END-PERFORM.
```

The Nested PERFORM

Throughout the first four chapters in the book we have seen examples of *nested* PERFORM statements: one paragraph performs another paragraph, which in turn performs another paragraph, and so on. Such nesting is common and useful. In a structure chart it corresponds to blocks at systematically lower levels. However, one important rule is that we cannot have cyclical (recursive) nesting. Consider this example:

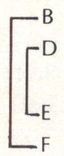
```
PAR-A.
    PERFORM PAR-B.
PART-B.
    PERFORM PAR-C.
PAR-C.
    PERFORM PAR-A
    PERFORM PAR-B.
```

Both PERFORM statements in PAR-C are in error; it is not valid to issue a PERFORM backwards. What will happen if we do? In general, the results are unpredictable, and often are very strange, depending on the program context and the particular compiler used. In fact, if a program compiles without errors and then during execution does "strange" things, it may be well worth the effort to check for the possibility of cyclical PERFORM statements.

When the THRU option is exercised, it is possible to have one or more PERFORM statements included in the range of procedure-1 to procedure-2. These are nested PERFORM statements. It is also possible to have PERFORM statements whose range is totally outside the range of the first PERFORM. The following two examples illustrate the two permitted structures:

CORRECT STRUCTURE

```
A. . . . PERFORM B THRU F.
B.
C. . . . PERFORM D THRU E.
D.
E.
F.
```



```
                        CORRECT STRUCTURE
A. . . PERFORM B THRU D.        ┌─B
B.                              │
                                │ . . . PERFORM M THRU Q.
C. . . PERFORM M THRU Q.        │
D.                              └─D
G.
M.                              ┌─M
I.                              │
Q.                              └─Q
```

Finally, the following example illustrates an incorrect structure:

```
                        INCORRECT STRUCTURE
A. . . . PERFORM B THRU E.      ┌─B
B.                              │┌─D
C. . . . PERFORM D THRU F.      ││
D.                              ││
E.                              └┼─E
F.                               └─F
```

Review

1 **The format of the PERFORM statement that specifies execution of a procedure a definite number of times is the PERFORM . . . ____________.**

TIMES

2 **The PERFORM . . . THRU format permits the execution of [two / two or more] procedures.**

two or more

3 **Suppose that four consecutive paragraphs are to be executed by use of the PERFORM . . . THRU command. The PERFORM statement specifically would include reference to [two / four] procedure-names.**

two

4 **Because a change in the physical position of paragraphs generally would disrupt the intent of a PERFORM . . . THRU command, the use of this format has to be considered as being ____________-prone.**

error

5 **When a PERFORM statement results in execution of a paragraph that itself contains another PERFORM statement (and so forth), the program structure is described as having a ____________ PERFORM structure.**

nested

6 **The type of nesting of PERFORM statements that is not allowed, and may have unpredictable results upon program execution, is ____________ nesting.**

cyclical (recursive)

THE GO TO VERB

For a long time, the GO TO was one of the most used verbs in all programming languages; however, it has been found that its unrestricted use is likely to create difficulty in reading a program and may contribute to the development of error-prone programs. In this book we refrain from the use of GO TO as much as possible; still, programs written in the first quarter-century of programming history are full of GO TO statements, and the reader should be familiar with this control verb.

GO TO is an unconditional branch to a paragraph (or section) name. Consider the following example which incorporates use of the GO TO:

```
READING-DATA.
    READ EMPLOYEE-FILE AT END GO TO WRAP-UP.
    PERFORM PROCESS-EMPLOYEE
    GO TO READING-DATA.
PROCESS-EMPLOYEE.
    .
    .
    .
WRAP-UP.
    PERFORM . . . .
    CLOSE . . .
    STOP RUN.
```

Notice that AT END the program specifies GO TO WRAP-UP, which presumably does the end-of-program processing. If it is not an end-of-file case, PERFORM PROCESS-EMPLOYEE is executed and then execution continues with GO TO READING-DATA. This last GO TO implements an iterative procedure, and it does so by referring to its own paragraph name.

Use of the GO TO in simple programs, such as the above example, seems straightforward and even, perhaps, appealing. However, in programs of typical size and complexity use of GO TO is likely to cause unnecessary complications in logic.

Review

1 The COBOL verb that provides the programmer with the capability of achieving unconditional branching in a program without returning to the point of original branching is ____________.

GO TO

2 The GO TO verb has been used in both COBOL and other programming languages. The use of this verb steadily has been [increasing / decreasing] in recent years.

decreasing

SAMPLE PROGRAM

We conclude this chapter by presenting a sample program to help the reader review many of the statements discussed as they apply to a particular application.

The objective of this program is to read records in a file and produce a report such as that shown in Figure 5-6. The input records are sorted so that all records pertaining to a given product are grouped together. The input records have the format presented in Figure 5-7. The report is relatively simple, but it is representative of the kind of report required for administrative use. In a more typical case, we would have several classes of data objects, such as department

```
PRODUCT NO.  SALES AMOUNT   TOTAL SALES

   123          125.27
                100.00
                              $225.27
   345           50.50
                110.25
                 10.09
                              $170.84
```

FIGURE 5-6
OUTPUT OF THE REPORT PROGRAM.

Record Format

PRODUCT-NO	SALES-AMOUNT	FILLER
PIC 999	PIC 9(4)V99	PIC X(71)

Sample Input

```
123012527
123010000
345005050
345011025
345001009
```

FIGURE 5-7
INPUT RECORD FORMAT AND SAMPLE RECORDS.

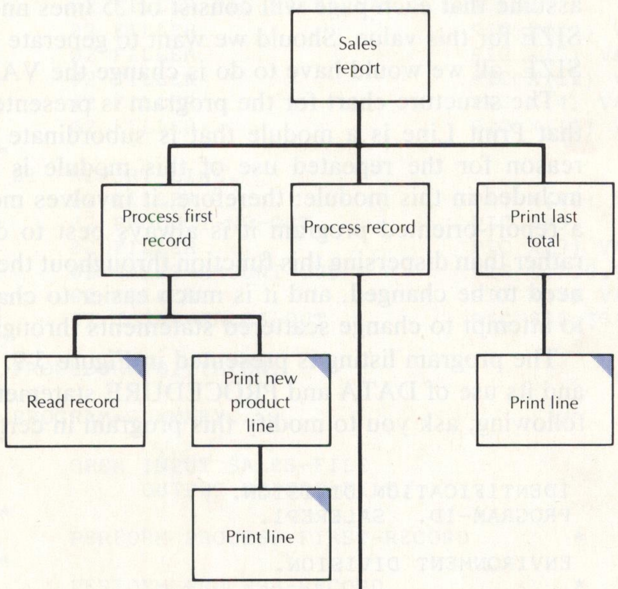


FIGURE 5-8
STRUCTURE CHART FOR THE REPORT PROGRAM.

and salesperson, in addition to the product classes; therefore we would want to present grouped data and totals for each class. For instance, we could have the sales for each product, for each salesperson, and for each department. We will illustrate such an expanded version of the current program in Chapter 9: "Sequential File Processing."

Now that we are at the stage of analyzing and designing such a program, we can observe several key points that are typical of such report-generating programs. First, the initial record that is input is treated separately. We have to check all subsequent records to ascertain whether or not the current record is for the same product as the previous record; this is not the case for the first record, for which there is no previous record. When the end of the file is reached, we should print the total sales for the product immediately preceding, before the program terminates. A final observation is that we will want to print a page heading at the top of each page. For this reason, it will be useful to develop a module whose function it will be to print each line and a page heading, as appropriate. We will assume that each page will consist of 25 lines and will define a constant PAGE-SIZE for this value. Should we want to generate the report in a different PAGE-SIZE, all we would have to do is change the VALUE clause of PAGE-SIZE.

The structure chart for the program is presented in Figure 5-8. It can be seen that Print Line is a module that is subordinate to five different modules. The reason for the repeated use of this module is that the page-heading logic is included in this module; therefore, it involves more than just printing a line. In a report-oriented program it is always best to do the printing in one module, rather than dispersing this function throughout the program. Report formats often need to be changed, and it is much easier to change one cohesive module than to attempt to change scattered statements throughout the program.

The program listing is presented in Figure 5-9. You should study the program and its use of DATA and PROCEDURE statements now. Exercises 5.3 and 5.4, following, ask you to modify this program in certain ways.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  SALEREP1.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
     SELECT SALES-FILE   ASSIGN TO  File-name/device.
     SELECT REPORT-FILE  ASSIGN TO  Printer.
/
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  SALES-FILE
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS SALES-RECORD.
 01  SALES-RECORD.
     02 PRODUCT-NO                      PIC 999.
     02 SALES-AMOUNT                    PIC 9(4)V99.
     02 FILLER                          PIC X(71).
*
 FD  REPORT-FILE
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS REPORT-RECORD.
 01  REPORT-RECORD                      PIC X(132).
```

FIGURE 5-9

LISTING OF THE REPORT PROGRAM.

```
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-INDICATOR         PIC XXX VALUE 'NO'.
     88 END-OF-FILE VALUE 'YES'.
     88 NOT-END-OF-FILE VALUE 'NO'.
*
 01  PREVIOUS-PRODUCT-NO           PIC 999.
*
 01  PRODUCT-TOTAL                 PIC 9(5)V99.
*
 01  PAGE-SIZE                     PIC 99 VALUE 25.
*
 01  PAGE-LINE-COUNTER                       PIC 99 VALUE 25.
*
*  BY INITIALIZING PAGE-LINE-COUNTER TO 25-PAGE-SIZE, WE
*  WILL GET THE PAGE HEADING PRINTED PRIOR TO THE FIRST
*  LINE OF OUTPUT.
*
 01  REPORT-HEADING.
     02 FILLER PIC X(10) VALUE SPACES.
     02 FILLER                     PIC X(11) VALUE 'PRODUCT NO.'.
     02 FILLER                     PIC X(3) VALUE SPACES.
     02 FILLER                     PIC X(12) VALUE 'SALES AMOUNT'.
     02 FILLER                     PIC X(4) VALUE SPACES.
     02 FILLER                     PIC X(11) VALUE 'TOTAL SALES'.
*
 01  REPORT-LINE.
     02 FILLER                     PIC X(14) VALUE SPACES.
     02 PRODUCT-NO-OUT             PIC 999.
     02 FILLER                     PIC X(9) VALUE SPACES.
     02 SALES-AMOUNT-OUT           PIC Z,ZZ9.99.
     02 FILLER                     PIC X(7) VALUE SPACES.
     02 TOTAL-SALES-OUT            PIC $$$,$$9.99.
/
 PROCEDURE DIVISION.
*
 PROGRAM-SUMMARY.
*
     OPEN INPUT SALES-FILE
          OUTPUT REPORT-FILE.
*
     PERFORM PROCESS-FIRST-RECORD
*
     PERFORM PROCESS-RECORD
             UNTIL END-OF-FILE
*
     PERFORM PRINT-LAST-TOTAL
*
     CLOSE SALES-FILE
           REPORT-FILE
*
     STOP RUN.
/
 PROCESS-FIRST-RECORD.
     PERFORM READ-RECORD.
     IF NOT-END-OF-FILE
        MOVE PRODUCT-NO TO PREVIOUS-PRODUCT-NO
        MOVE SALES-AMOUNT TO PRODUCT-TOTAL
     PERFORM PRINT-NEW-PRODUCT-LINE
     PERFORM READ-RECORD.
*
 READ-RECORD.
     READ SALES-FILE
          AT END MOVE 'YES' TO END-OF-FILE-INDICATOR.
*
 PRINT-NEW-PRODUCT-LINE.
     MOVE SPACES TO REPORT-LINE
     MOVE PRODUCT-NO TO PRODUCT-NO-OUT
     MOVE SALES-AMOUNT TO SALES-AMOUNT-OUT
     PERFORM PRINT-LINE.
```

FIGURE 5-9 *(Continued)*
LISTING OF THE REPORT PROGRAM.

```
PRINT-LINE.
    IF PAGE-LINE-COUNTER = PAGE-SIZE
       WRITE REPORT-RECORD FROM REPORT-HEADING
             AFTER ADVANCING PAGE
       MOVE SPACES TO REPORT-RECORD
       WRITE REPORT-RECORD AFTER ADVANCING 1 LINE
       MOVE 2 TO PAGE-LINE-COUNTER.
    WRITE REPORT-RECORD FROM REPORT-LINE
          AFTER ADVANCING 1 LINE
    ADD 1 TO PAGE-LINE-COUNTER.
/
PROCESS-RECORD.
    IF PRODUCT-NO = PREVIOUS-PRODUCT-NO
       PERFORM PROCESS-REPORT-LINE
    ELSE
       PERFORM PROCESS-NEW-PRODUCT.
*
    ADD SALES-AMOUNT TO PRODUCT-TOTAL.
*
    PERFORM READ-RECORD.
*
PROCESS-REPORT-LINE.
    MOVE SPACES TO REPORT-LINE
    MOVE SALES-AMOUNT TO SALES-AMOUNT-OUT
    PERFORM PRINT-LINE.
*
PROCESS-NEW-PRODUCT.
    MOVE SPACES TO REPORT-LINE
    MOVE PRODUCT-TOTAL TO TOTAL-SALES-OUT
    PERFORM PRINT-LINE
*
    MOVE ZERO TO PRODUCT-TOTAL
*
    PERFORM PRINT-NEW-PRODUCT-LINE
*
    MOVE PRODUCT-NO TO PREVIOUS-PRODUCT-NO.
*
PRINT-LAST-TOTAL.
    MOVE SPACES TO REPORT-LINE
    MOVE PRODUCT-TOTAL TO TOTAL-SALES-OUT
    PERFORM PRINT-LINE.
```

FIGURE 5-9 *(Continued)*
LISTING OF THE REPORT PROGRAM.

EXERCISES

5.1 Write a COBOL program that will read input records, edit the data, and print a report as indicated below.

Input:

COLUMNS	CONTENT
1–11	Social security number
12–23	Last name
24–32	First name
33–40	Blank
41–47	Year-to-date earnings

Output: not more than a page. Header at the top of the page, as follows:

SOCIAL SECURITY	LAST	FIRST	EARNINGS
XXX-XX-XXXX			$XX,XXX.XX
XXX-XX-XXXX			$XX,XXX.XX
	(etc.)		

Data: create four records using social security numbers and names of your choice. However, use these earnings:

$10,352.81
$ 5,863.98
$ 3,800.00
$ 691.12

5.2 Write the PROCEDURE DIVISION statements to produce the output described in Exercise 4.7.

5.3 Modify the sample program listed in Figure 5-9 so that a grand total of all sales amounts is printed on a separate page at the end of the report, with the following format:

GRAND-TOTAL FOR SALES REPORT = $XXX,XXX.XX

5.4 Modify the sample program listed in Figure 5-9 so that at the end of the report a summary is given of amounts of sales, with the following format:

PERCENT OF SALES LESS THAN $100	PERCENT OF SALES $100–$500	PERCENT OF SALES OVER $500
XXX.X	XXX.X	XXX.X

AVERAGE VALUE OF SALES ORDER = $X,XXX.XX

In other words, the program should tabulate the SALES-AMOUNT field so that the required percentage amounts can be computed. The program should determine what percent of the records had a SALES-AMOUNT less than $100, what percent had an amount in the range of $100–$500, and what percent exceeded $500. The final average is the sum of SALES-AMOUNT values divided by the number of records.

5.5 Write a COBOL program to read a file of records containing data about accounts-receivable and to print a summary of the overdue and forthcoming receivables. An extended description follows, including variable names and a partial DATA DIVISION.

INPUT

This consists of a set of records having the following design:

COLUMNS	COBOL NAME
1–6	ACCOUNT-NO
7–8	FILLER (blank)
9–11	DAY-DUE†
12–13	YEAR-DUE
14–21	AMOUNT-DUE
22–80	FILLER (blank)

†The day on which the account is due is expressed as a 3-digit number (Julian calendar). Thus January 10 is 010 and February 28 is 059.

Use the appropriate form of ACCEPT to access today's date.

The records are sorted in ascending sequence on ACCOUNT-NO, and there should be only one record per account. The program checks for correct sequencing. Records out of sequence are to be printed as shown on the sample output. Note that the data on such records are excluded from the total.

DATA DIVISION

All DATA DIVISION entries are provided, except that you are asked to write WORKING-STORAGE entries to provide the header with the words STATUS, NUMBER OF ACCOUNTS, and DOLLAR VALUE.

```
DATA DIVISION.
FILE SECTION.
FD  REC-FILE LABEL RECORDS OMITTED
    DATA RECORD RECEIV-RECORD.
01  RECEIV-RECORD.
    02  ACCOUNT-NO              PICTURE 9(6).
    02  FILLER                  PICTURE XX.
    02  DAY-DUE                 PICTURE 999.
    02  YEAR-DUE                PICTURE 99.
    02  AMOUNT-DUE              PICTURE 9(6)V99.
    02  FILLER                  PICTURE X(59).
FD  REPORT-FILE LABEL RECORD OMITTED
    DATA RECORD REPORT-RECORD.
01  REPORT-RECORD               PICTURE X(132).
WORKING-STORAGE SECTION.
01  TODAY                       PICTURE 999.
01  THIS-YEAR                   PICTURE 99.
01  PREVIOUS-ACCT-NO            PICTURE 9(6).
01  OVERDUE                     PICTURE S9(6).
01  OVERDUE-AMOUNT              PICTURE S9(6)V99.
01  RECEIVABLE-AMOUNT           PICTURE S9(6)V99.
01  END-OF-DATA                 PICTURE 9 VALUE ZERO.
01  ERROR-MESSAGE.
    02  FILLER                  PICTURE X(23) VALUE
                                'ACCOUNT OUT OF SEQUENCE.'
    02  FILLER                  PICTURE XX VALUE SPACE.
    02  ERROR-NUMBER            PICTURE X(6).
01  RESULT.
    02  STATUS-TYPE             PICTURE X(13).
    02  FILLER                  PICTURE X(9) VALUE SPACES.
    02  HOW-MANY                PICTURE ZZZ99.
    02  FILLER                  PICTURE X(13), VALUE SPACES.
    02  DOLLAR-VALUE            PICTURE $$$$,$$$.99.
```

A few data-names merit explanation:

TODAY Stores today's data as ACCEPTed by your program using additional fields

THIS-YEAR Stores the current year as accessed by the ACCEPT verb.

PREVIOUS-ACCT-NO Stores the previously read account number so that each record can be compared with the preceding one to see that they are in ascending sequence. Initally, PREVIOUS-ACCT-NO is set equal to zero

OVERDUE Stores the number of overdue accounts

OVERDUE-AMOUNT Stores the total dollar value of overdue accounts

RECEIVABLE Stores the number of accounts that are not overdue

RECEIVABLE-AMOUNT Stores the total value of accounts that are not overdue

SAMPLE INPUT for day 101 of 1987

012345	0908700010000
023467	1508600020020
001234	1408600030030
123456	0308800040040

SAMPLE OUTPUT

ACCOUNT OUT OF SEQUENCE 001234

STATUS	NUMBER OF ACCOUNTS	DOLLAR VALUE
OVERDUE	02	$300.20
RECEIVABLE	01	$400.40
TOTALS	03	$700.60

5.6 A record file contains data on the number of vehicles passing a certain point on the highway. Each record contains a whole number in columns 1–4 that corresponds to one day's traffic. We are interested in reading the data file and producing a report such as that shown below. For each week, we list the daily data and a statistical summary for the week. It is assumed that, at the most, there will be 10 weeks of data; however, the last week of data may be a partial week—that is, less than seven days. The output format is as follows:

```
DAILY TRAFFIC REPORT
    WEEK XX
    X, XXX
    X, XXX
    .
    .
    .
    .
    X, XXX
MINIMUM =X,XXX  MAXIMUM =X,XXX  WEEKLY TOTAL = XX,XXX
```

6 CONDITIONS AND CONDITIONAL STATEMENTS

INTRODUCTION

Computer processors derive their logic capability from their ability to test for the truth or falsity of conditions. We can appreciate the fundamental nature of this ability by considering what computer programs would be like without conditional logic. It would be impossible, in fact, to write most computer programs without the use of conditional instructions. The COBOL language recognizes the importance of this capability and provides a rich repertoire of conditions and conditional statements. We present the subject as a separate chapter because of the number of conditions that can be tested and because we wish to highlight the logical unity of various types of conditional statements.

We have already studied a number of conditional statements in preceding chapters. In this chapter, we expand on the subject of conditions and conditional statements by discussing six types of conditions: (1) relation conditions, (2) nested conditions, (3) class conditions, (4) sign conditions, (5) condition-name conditions, and (6) complex conditions. In general, a condition is an expression that is either true or false in a particular circumstance; that is, the condition either holds or it does not hold.

RELATION CONDITIONS

Such conditions are concerned with comparisons between two items. The type of comparison is indicated by the relational operator, which may be in the form of words or symbols. The relational operators available in COBOL are included in the following general format:

```
                                 ⎧ IS [NOT] LESS THAN      ⎫
                                 ⎪ IS [NOT] <              ⎪
                                 ⎪ IS [NOT] EQUAL TO       ⎪
                                 ⎪ IS [NOT] =              ⎪
   ⎧ identifier-1              ⎫ ⎪ IS [NOT] GREATER THAN;  ⎪ ⎧ identifier-2              ⎫
IF ⎨ literal-1                 ⎬ ⎨ IS [NOT] >              ⎬ ⎨ literal-2                 ⎬
   ⎩ arithmetic-expression-1   ⎭ ⎪ IS GREATER THAN OR      ⎪ ⎩ arithmetic-expression-2   ⎭
                                 ⎪    EQUAL TO             ⎪
                                 ⎪ IS > =                  ⎪
                                 ⎪ IS LESS THAN OR         ⎪
                                 ⎪    EQUAL TO             ⎪
                                 ⎩ IS < =                  ⎭
```

Examples of the use of the above format are:

IF AMOUNT IS LESS THAN 100
IF AMOUNT < 100
IF BALANCE IS NOT EQUAL TO CREDIT-LIMIT
IF BALANCE IS NOT = CREDIT-LIMIT
IF HOURS − 40 > ZERO
IF QUANTITY GREATER OR EQUAL MAX-SIZE
IF QUANTITY > = MAX-SIZE

As the above examples illustrate, one can use either the symbolic (<, =, >) or the verbal (LESS, EQUAL, GREATER) operators; they are equivalent forms, and it is a matter of personal preference as to which is used.

The fifth example illustrates use of an arithmetic expression (HOURS − 40). Rules for forming and evaluating such arithmetic expressions were discussed in Chapter 5, in connection with the COMPUTE arithmetic verb.

In the general format just presented, the first and second operands often are referred to as the subject and object of the condition, respectively. One additional point in this regard is that these two operands cannot both be literals. Thus, the statement IF 3 IS GREATER THAN 2 is invalid. Of course, there is no reason why such a statement ever need be written, since the equality or inequality is predetermined by the statement itself and does not constitute an actual relational comparison.

The meaning of relational tests involving numeric values is obvious. If RATE-1 and RATE-2 are numeric fields, the conditional expression IF RATE-1 > RATE-2 PERFORM PAR-A leads to PAR-A whenever the numeric (algebraic) value of RATE-1 is greater than the numeric value of RATE-2, *regardless of the field size*.

With respect to nonnumeric comparisons, however, certain things are not so obvious. A comparison involving two alphabetic items proceeds from left to right in pairs of characters until the first unequal pair occurs. Thus, in comparing |T|H|O|R|P| to |T|H|A|L|E|S|, the first field is determined to be greater than the second where the O-A pair is compared. The size of the fields in this case is irrelevant, but suppose we are comparing the following two fields:

|S|A|N|D|E|R|S| |S|A|N|D|E|R|S|O|N|

In this case the first field is considered to be smaller in value. The shorter field is treated as if it had enough trailing blanks to make it equal in size to the longer field. Then, when a blank is compared to the letter "O" in the eighth field position, the first field is determined to be smaller in value.

As a third case of alphabetic comparison, consider this:

S	M	I	T	H

S	M	I	T	H		

The two items are considered equal, even though their field size is unequal. In this case, the first field is treated as if it had enough trailing blanks to be equal in size to the second field.

Consider the following two alphanumeric fields:

X	A	-	1	1	4

X	.	B	-	3

Convention for determining the "larger" of the two does not exist; still, these are legitimate fields in computer processing, and we must handle them. In this respect, we rely on the *collating sequence* of a computer system. The collating sequence is simply a defined sequence that states the relative "size" of each possible character in a computer's storage. For example, in the widely used IBM large mainframe computers, the EBCDIC collating sequence for the following characters is as indicated by the order of listing:

blank
. (period or decimal point)
(
+ (plus sign)
)
- (hyphen or minus)
/ (stroke)
, (comma)
′ (single quote)
″ (double quote)
letters A–Z
numbers 0–9

Thus, when comparing 3-A/ and Z/9K, the 3-A/ is the greater.

There are several collating sequences in use, and one should be aware of the possible differences in program results arising from differences in collating sequences. The standard one is the American Standard Code for Information Interchange (ASCII).

Review

1 Relation conditions are concerned with comparisons between two items. The words or symbols that serve to indicate the type of comparison to be made are called relational ________.

operators

2 As indicated above, relational operators can be in the form of either words or symbols. In the following spaces, enter the symbols that are equivalent to the listed relational operators:
LESS THAN ________
EQUAL TO ________
GREATER THAN ________

< = >

3 Of the following three relation conditions, the one that is invalid as a COBOL expression is the one identified by the letter **[a / b / c]**.

a IF GROSS-PAY IS GREATER THAN 99 . . .
b IF 100 < ORDER-AMT . . .
c IF 500 > 400 . . .

c

4 An ordering that defines the relative rank of all the valid characters in a computer system is referred to as the ________________ for the system.

collating sequence

NESTED CONDITIONS

Before discussing nested conditions, let us consider the general format for the IF statement.

```
                 ⎧ statement-a    ⎫ ⎧ ELSE {statement-2} . . . [END-IF] ⎫
IF condition THEN⎨                ⎬ ⎨ ELSE NEXT SENTENCE                ⎬
                 ⎩ NEXT SENTENCE  ⎭ ⎩ END-IF                            ⎭
```

The possible unique structures are many, and, since the IF statement is a very common and very useful program instruction, we will do well to spend some time studying it. Here are some of the forms we can have:

IF condition statement-1.
IF condition statement-1 ELSE NEXT SENTENCE.
IF condition statement-1 ELSE statement-2.
IF condition NEXT SENTENCE ELSE statement-2.

Note that the first case shows that we can omit the ELSE portion. For example, we can write IF SUM-TAX < 50.00 MOVE ZERO TO DEDUCTIONS. The period following DEDUCTIONS signifies that the current sentence has ended. Thus, the statement that follows will be the NEXT SENTENCE, and so it would be redundant to write ELSE NEXT SENTENCE, although it would not be wrong.

To enhance your understanding of conditional statements, study the following example:

```
IF AMOUNT IS GREATER THAN CREDIT LIMIT
    WRITE PRINT-LINE FROM CREDIT-OVERDRAW
ELSE
    MOVE AMOUNT TO BILLING-FIELD
    WRITE PRINT-LINE FROM BILL-AREA.
ADD AMOUNT TO TOTAL-VALUE.
```

The flowchart for this program segment appears in Figure 6-1. The program statements that correspond to the flowchart descriptions are as follows:

FLOWCHART DESCRIPTION	CORRESPONDING PROGRAM STATEMENT
Condition	IF AMOUNT IS GREATER THAN CREDIT-LIMIT
Statement-1	WRITE PRINT LINE FROM CREDIT-OVERDRAW
Statement-2	MOVE AMOUNT TO BILLING-FIELD WRITE PRINT-LINE FROM BILL-AREA
Next sentence	ADD AMOUNT TO TOTAL-VALUE

Notice that statement-1 and statement-2 need not be single statements. Statement-2 illustrates the case in which two statements are included.

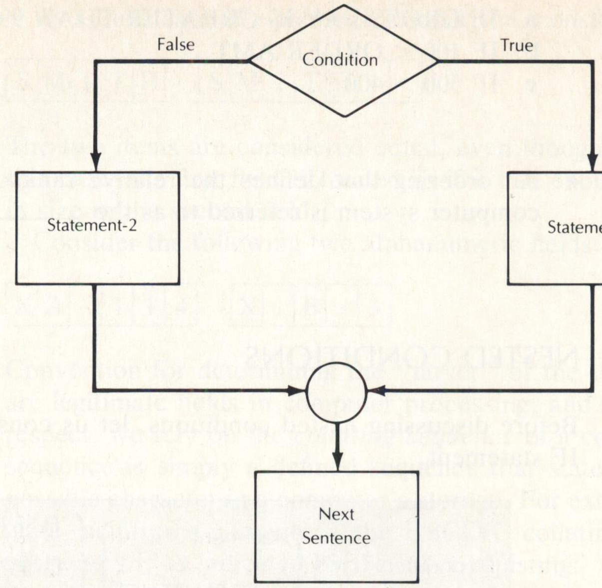


FIGURE 6-1
FLOWCHART FOR THE EXAMPLE OF A CONDITIONAL STATEMENT.

The NEXT SENTENCE option is one way of expressing the null leg of a condition, as in this example:

```
IF DEBITS = CREDITS
    NEXT SENTENCE
ELSE
    PERFORM OUT-OF-BALANCE.
```

The NEXT SENTENCE provides a clear way of expressing the fact that no action is taken if the condition is true. Contrast the example just given to the equivalent:

```
IF DEBITS NOT = CREDITS
    PERFORM OUT-OF-BALANCE.
```

For most people, the first version is clearer.

In the general format for the IF statement, statement-1 and statement-2 are not restricted to being imperative statements; rather, they themselves may be conditional expressions, giving rise to the possibility of using nested IF statements. A relatively simple example of a nested IF statement, or a nested conditional, is the following:

```
IF AMOUNT IS LESS THAN 100
    IF AMOUNT IS GREATER THAN 50
        MOVE 0.3 TO RATE
    ELSE
        MOVE 0.4 TO RATE
ELSE
    MOVE 0.2 TO RATE
```

This COBOL statement corresponds to the following rule:

AMOUNT	RATE
Less than or equal to 50	0.4
Greater than 50 but less than 100	0.3
Equal to or greater than 100	0.2

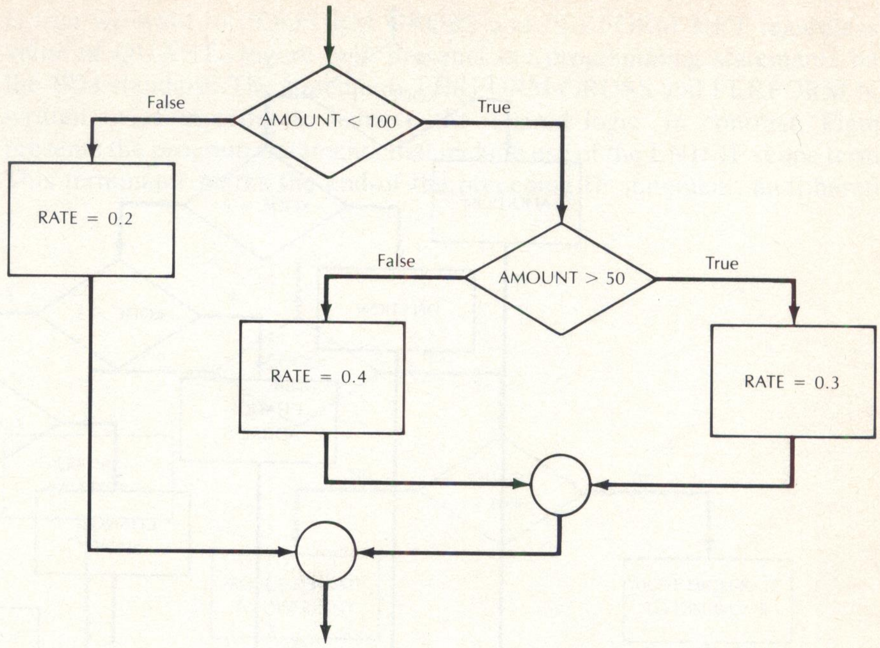


FIGURE 6-2
NESTED CONDITIONAL STRUCTURE TO DETERMINE RATE.

You will find it useful in interpreting nested conditionals to look for the first ELSE; it always pertains to the immediately preceding IF. The second ELSE pertains to the IF just preceding the inner IF, and so on. Schematically, the relationships can be portrayed as follows:

```
IF . . . IF . . . IF . . . ELSE . . . ELSE . . . ELSE . . .
```

The flowchart for this example is shown in Figure 6-2.

Nested conditions can be very useful in writing program statements, but they also can be misused. This is done by nesting conditions so deeply that program logic is not easy to follow. Because nested conditions are important, we present an additional example.

Consider a code that can have legitimate values in the range 1–5. Figure 6-3 depicts a nested conditional structure that tests for the value of the code and executes a suitable procedure. In language form, we could write:

```
IF KODE = 1
    PERFORM ADDITION
ELSE
    IF KODE = 2
        PERFORM DELETION
    ELSE
        IF KODE = 3
            PERFORM CHANGE-ADDRESS
        ELSE
            IF KODE = 4
                PERFORM CHANGE-NAME
            ELSE
                IF KODE = 5
                    PERFORM CHANGE-CREDIT
                ELSE
                    PERFORM ERROR-CODE.
PERFORM NEXT-P.
```

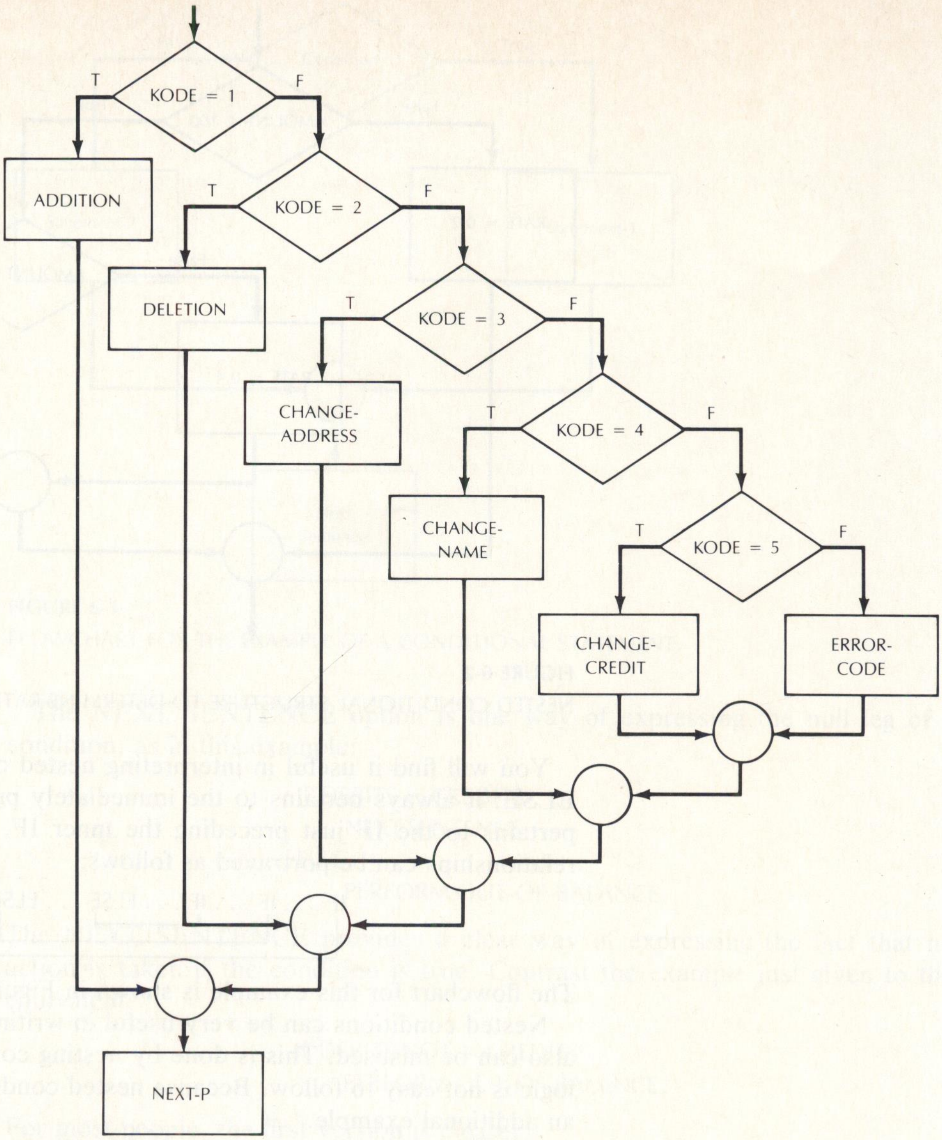


FIGURE 6-3
SAMPLE NESTED CONDITIONAL STRUCTURE THAT INCLUDES FIVE LEVELS.

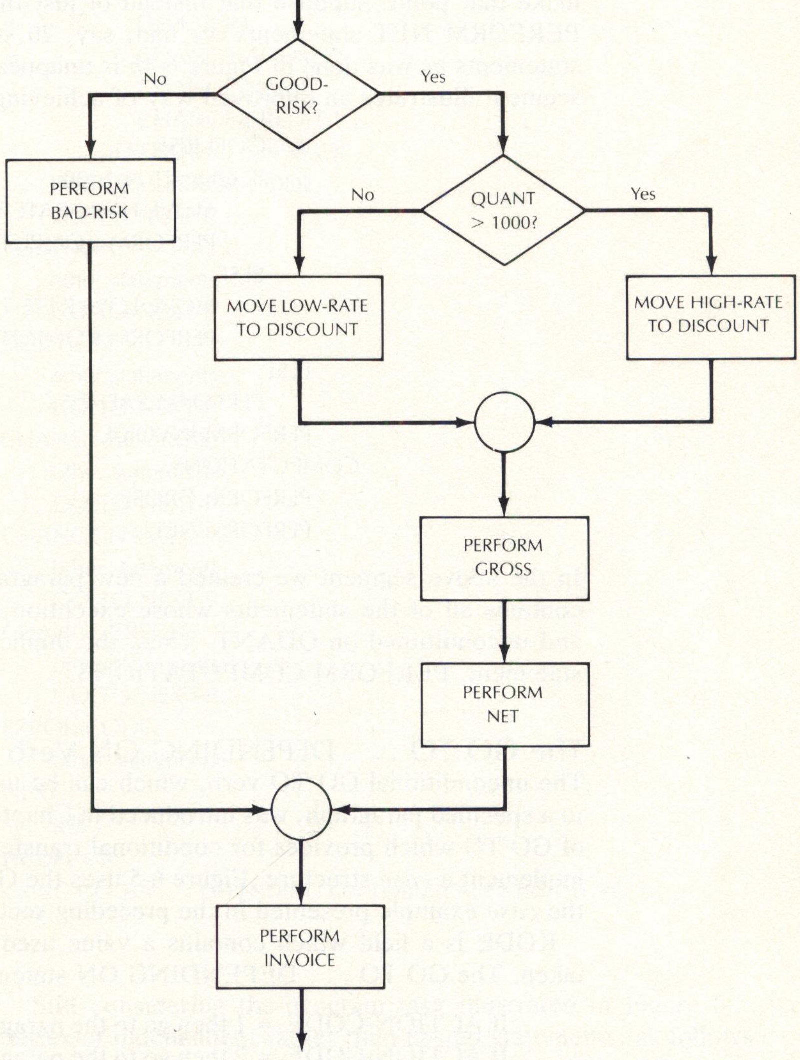
We have nested to five levels, which tests the limits of our ability to understand the program logic inherent in the nesting. In general, many programming managers advise against nesting more than three levels. In this particular example, the structure is rather easy, however, because of the null alternatives involved. In this sense, we can say that, even though we have nested to five levels, it is a "clean" program structure.

With respect to the above example, recall the *case* structure which was presented in Figure 3-4*f* in Chapter 3. One way of implementing the case structure is by means of nested IF statements. There are also other ways of implementing the case structure, and these will be discussed in subsequent sections of this chapter in connection with the GO TO . . . DEPENDING ON and the EVALUATE statements.

The END-IF Scope Terminator

Nested structures are sometimes a bit awkward to code in the 1974 version of COBOL. Consider the flowchart in Figure 6-4*a*. When the GOOD-RISK condition

is true we want to PERFORM GROSS and PERFORM NET regardless of the value of QUANT. Figure 6-4*b* presents the programming statements based on the 1974 standard. The statements PERFORM GROSS and PERFORM NET are written twice in order to achieve the desired logic. In contrast, Figure 6-4*c* presents the program statements that include use of the END-IF scope terminator. This terminator marks the end of the preceding IF statement, and therefore the



(a) Sample Nested Structure

```
IF GOOD-RISK
   IF QUANT > 1000
      MOVE HIGH-RATE TO DISCOUNT
      PERFORM GROSS
      PERFORM NET
   ELSE
      MOVE LOW-RATE TO DISCOUNT
      PERFORM GROSS
      PERFORM NET
ELSE
   PERFORM BAD-RISK.
PERFORM INVOICE.
```

(b) Without Use of END-IF

```
IF GOOD-RISK
   IF QUANT > 1000
      MOVE HIGH-RATE TO DISCOUNT
   ELSE
      MOVE LOW-RATE TO DISCOUNT
   END-IF
   PERFORM GROSS
   PERFORM NET
ELSE
   PERFORM BAD-RISK
END-IF
PERFORM INVOICE.
```

(c) With Use of END-IF

FIGURE 6-4

ILLUSTRATION OF NESTED IF WITH AND WITHOUT END-IF.

PERFORM GROSS and PERFORM NET are executed conditionally on GOOD-RISK being true and *unconditionally* on the value of QUANT. A second END-IF is used just before the PERFORM INVOICE statement. A period would play the same role as the second END-IF because a period would also terminate the scope of the main IF GOOD-RISK statement. The example in Figure 6-4*c* also illustrates use of the optional THEN keyword.

While the END-IF plays a useful role, one should not conclude that the 1974 standard need be deficient in ease and clarity of program structure. In order to make that point, suppose that instead of just the *two* PERFORM GROSS and PERFORM NET statements we had, say, 20 statements. Duplicating all such statements as was done in Figure 6-4*b* is unappealing, but the following program segment illustrates an improved way of achieving the desired result:

```
    IF GOOD-RISK
        IF QUANT > 1000
            MOVE HIGH-RATE TO DISCOUNT
            PERFORM COMPUTATIONS
        ELSE
            MOVE LOW-RATE TO DISCOUNT
            PERFORM COMPUTATIONS
    ELSE
        PERFORM BAD-RISK.
    PERFORM INVOICE.
COMPUTATIONS.
    PERFORM GROSS
    PERFORM NET.
```

In the above segment we created a new paragraph, COMPUTATIONS, which contains all of the statements whose execution is conditional on GOOD-RISK and unconditional on QUANT. Thus, the duplicated code consists of only one statement, PERFORM COMPUTATIONS.

The GO TO . . . DEPENDING ON Verb

The unconditional GO TO verb, which can be used to transfer program control to a specified paragraph, was introduced in Chapter 5. There is also another form of GO TO which provides for conditional transfer of control and can be used to implement a *case* structure. Figure 6-5 uses the GO TO as the basis for rewriting the *case* example presented in the preceding section.

KODE is a field which contains a value used to determine the action to be taken. The GO TO . . . DEPENDING ON statement says, in effect:

If ACTION-CODE = 1 then go to the paragraph named ADDITION
If ACTION-CODE = 2 then go to the paragraph named DELETION
(etc.)

There are five paragraph-names presented between GO TO and DEPENDING; therefore the command specifies what is to be done if KODE has values of 1, 2, 3, 4, or 5. If KODE has some other value, then the next statement (PERFORM ERROR-CODE) is executed. If three paragraphs had been named instead of five, then conditional branching would be specified for three values of KODE, namely, values of 1, 2, and 3. Thus, the identifier following the DEPENDING ON clause is tested for values of 1, 2, . . . up to the number of paragraph names used.

Reviewing Figure 6-5, we note that after the conditional branch to one of the five named paragraphs we used GO TO NEXT-P as the last statement in each paragraph. In this way we preserve the one-entry one-exit principle of good programming discussed in Chapter 3. Once execution of the paragraph named SELECT-ACTION begins, we are sure to end at NEXT-P.

```
01  KODE PIC 9.
    88  ADD-RECORD VALUE 1.
    88  DELETE-RECORD VALUE 2.
    88  ADDRESS-CHANGE VALUE 3.
    88  NAME-CHANGE VALUE 4.
    88  CREDIT-CHANGE VALUE 5.
    88  CODE-ERROR VALUES ARE 0 6 THRU 9.
SELECT-ACTION.
    GO TO ADDITION
          DELETION
          CHANGE-ADDRESS
          CHANGE-NAME
          CHANGE-CREDIT
       DEPENDING ON KODE
    PERFORM ERROR-CODE
    GO TO NEXT-P.
ADDITION.
    other statements
    GO TO NEXT-P.
DELETION.
    other statements
    GO TO NEXT-P.
CHANGE-ADDRESS.
    other statements
    GO TO NEXT-P.
CHANGE-NAME.
    other statements
    GO TO NEXT-P.
CHANGE-CREDIT.
    other statements
    GO TO NEXT-P.
ERROR-CODE.
    other statements
NEXT-P.
    (etc.)
```

FIGURE 6-5
USING GO TO . . . DEPENDING ON FOR A CASE STRUCTURE.

Still considering the program task illustrated in Figure 6-5, we could use a series of independent rather than nested statements, as follows:

```
IF ADD-RECORD          PERFORM ADDITION.
IF DELETE-RECORD       PERFORM DELETION.
IF ADDRESS-CHANGE      PERFORM CHANGE-ADDRESS.
IF NAME-CHANGE         PERFORM CHANGE-NAME.
IF CREDIT-CHANGE       PERFORM CHANGE-CREDIT.
IF CODE-ERROR          PERFORM ERROR-CODE.
PERFORM NEXT-P.
```

In this case, the programming is simpler, but there are two minor disadvantages. First, it is not immediately obvious that only one of these seven conditions can be true, and therefore program clarity suffers. Second, all seven conditions will be tested before the PERFORM NEXT-P is executed, whereas in the nested and GO TO structures, as soon as one condition holds, control moves immediately to PERFORM NEXT-P.

The EVALUATE Verb

The revised language provides for yet another way to implement *case* structures, by use of the EVALUATE verb. We introduce the verb by applying it to the same example used in the preceding sections of the chapter.

```
EVALUATE KODE
    WHEN 1  PERFORM  ADDITION
    WHEN 2  PERFORM  DELETION
    WHEN 3  PERFORM  CHANGE-ADDRESS
    WHEN 4  PERFORM  CHANGE-NAME
    WHEN 5  PERFORM  CHANGE-CREDIT
    WHEN OTHER  PERFORM  ERROR-CODE
END-EVALUATE
PERFORM NEXT-P.
```

As the statement clearly indicates, when KODE = 1, we PERFORM ADDITION, and so on. The WHEN OTHER option specifies the "catchall" category. If KODE is not 1, 2, 3, 4, or 5 then we PERFORM ERROR-CODE. The END-EVALUATE is the scope terminator, so that we clearly mark the end of the EVALUATE statement. Of course, a period would have the same effect if it were used in place of END-EVALUATE but would not be as clear conceptually.

An alternate formulation of the example could use the TRUE option of the EVALUATE statement and the 88-level condition-names defined earlier for KODE, as follows:

```
EVALUATE TRUE
    WHEN  ADD-RECORD       PERFORM  ADDITION
    WHEN  DELETE-RECORD    PERFORM  DELETION
    WHEN  ADDRESS-CHANGE   PERFORM  CHANGE-ADDRESS
    WHEN  NAME-CHANGE      PERFORM  CHANGE-NAME
    WHEN  CREDIT-CHANGE    PERFORM  CHANGE-CREDIT
    WHEN  OTHER            PERFORM  ERROR-CODE
END-EVALUATE
PERFORM NEXT-P.
```

The EVALUATE can also be used in more complex ways, such as to specify the conditions under which individual branches of the *case* construct are selected. Figure 6-6 presents a simple decision table that shows two fields, YEAR-CODE and LETTER-GRADE. There are four actions specified depending on the joint values of the two fields. For instance, when YEAR-CODE has a value of 4 we execute the procedure called PROC-3 regardless of the value of LETTER-GRADE. Figure 6-6 also includes an EVALUATE statement to implement the decision table. The ALSO has the obvious meaning. For example, the first WHEN clause is interpreted as follows:

> If the value of YEAR-CODE is 1 thru 2 and the value of
> LETTER-GRADE is A thru C, then PERFORM PROC-1.

The ANY option in the third WHEN in Figure 6-6 specifies that any value of LETTER-GRADE fulfills the evaluation criterion for this WHEN conditional. Finally, the WHEN OTHER refers to both the YEAR-CODE and LETTER-GRADE; it is equivalent to saying:

> Any other combination of values of YEAR-CODE and LETTER-GRADE,
> other than the ones specified in the previous WHEN's.

The complete format of the EVALUATE verb is included in Appendix B. As can be seen in the appendix, it is a complex statement with a great many options that go beyond the scope of the present chapter. Thus, we present a reduced

CONDITION		
YEAR-CODE	LETTER-GRADE	ACTION
1 or 2	A, B, or C	execute PROC-1
3	A or B	execute PROC-2
4	any	execute PROC-3
any other	any	execute PROC-4

```
EVALUATE       YEAR-CODE       ALSO        LETTER-GRADE
      WHEN     1 THRU 2    ALSO 'A' THRU 'C'   PERFORM PROC-1
      WHEN     3           ALSO 'A' THRU 'B'   PERFORM PROC-2
      WHEN     4           ALSO ANY            PERFORM PROC-3
    WHEN OTHER PERFORM PROC-4
END-EVALUATE.
```

FIGURE 6-6
SAMPLE USE OF EVALUATE VERB.

```
EVALUATE   {identifier-1}       [ALSO identifier-2] . . .
           {TRUE        }

  {        {ANY                                                              }
  {        {condition-1                                                      }
  { WHEN   {                                                                 }
  {        {[NOT] {identifier-3} [{THROUGH} {identifier-4}]                  }
  {        {      {literal-1   } [{THRU   } {literal-2   }]                  }

  [        {ANY                                                              ]
  [        {condition-2                                                      ]
  [ ALSO   {                                                                 ]
  [        {[NOT] {identifier-5} [{THROUGH} {identifier-6}]                  ]
  [        {      {literal-3   } [{THRU   } {literal-4   }]                  ]

  imperative-statement-1 } . . .

  [WHEN OTHER imperative-statement-2]

  [END-EVALUATE]
```

FIGURE 6-7
PARTIAL FORMAT FOR EVALUATE STATEMENT.

format that can be used for typical programming needs. Figure 6-7 presents the reduced format, and the two examples presented above illustrate most of the options in the format. In our examples we used literals rather than identifiers. If identifiers are used, then their content is the basis for evaluation. For example, consider these statements:

```
EVALUATE KODE
      WHEN ADD-CODE PERFORM ADD-PAR . . .
```

The above statement in effect says to perform ADD-PAR if the content of KODE is equal to the content of ADD-CODE (a field assumed to be relevant for the task).

Condition-1, condition-2, etc., in the format in Figure 6-7 refer to conditional expressions, such as condition-names, relational conditions, class conditions, and so on.

Review

1 Identify, for the following program instruction, the program statement that corresponds to each IF statement element. Refer to the general format for the IF statement if you wish.

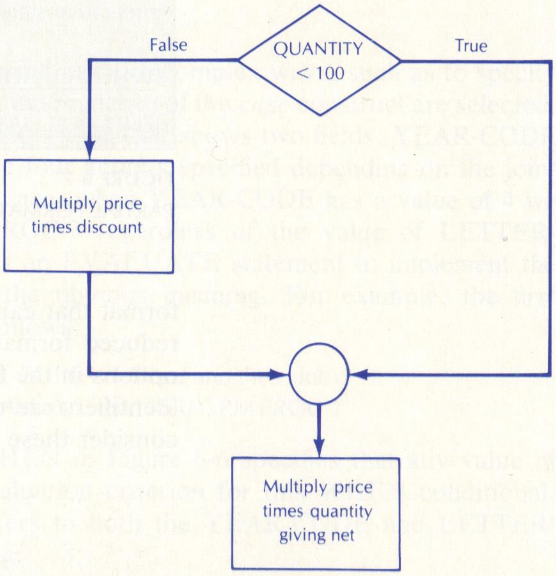
```
IF  QUANTITY < 100
    NEXT SENTENCE
ELSE
    MULTIPLY DISCOUNT BY PRICE.
MULTIPLY PRICE BY QUANTITY GIVING NET.
```

IF STATEMENT ELEMENT	CORRESPONDING PROGRAM STATEMENT
a Condition	__________
b Statement-1	__________
c Statement-2	__________
d Next Sentence	__________

a QUANTITY < 100
b *Not used*
c MULTIPLY DISCOUNT BY PRICE
d MULTIPLY PRICE BY QUANTITY GIVING NET

2 Construct a flowchart for this program segment:

```
IF  QUANTITY < 100
    NEXT SENTENCE
ELSE
    MULTIPLY DISCOUNT BY PRICE.
MULTIPLY PRICE BY QUANTITY GIVING NET.
```



3 Using the following COBOL statement, complete the table below, summarizing the decision rule being used:

```
IF  GRSPAY < 1000.00
    IF  GRSPAY > 500.00
        MOVE 0.05 TO RETRMNT-DEDUC
```

```
ELSE
    MOVE 0.03 TO RETRMNT-DEDUC
ELSE
    MOVE 0.07 TO RETRMNT-DEDUC.
```

AMOUNT OF GROSS PAY	RETIREMENT DEDUCTION RATE
Equal to or greater than 1,000	______
Greater than 500 but less than 1,000	______
Less than or equal to 500	______

0.07; 0.05; 0.03

4 The type of conditional statement illustrated in the preceding question usually is referred to as a(n) ______________ conditional.

nested

5 The logic associated with a deeply nested program structure can be clarified by decomposition of the program into several separate, but dependent, ______________.

modules (or paragraphs)

6 With respect to the types of program structure presented in Chapter 3, nested IF statements can be used to implement the ______________ type of program structure.

case

7 The END-IF scope terminator available with the revised language permits us to implement nested structures in COBOL programs [more clearly / more extensively].

more clearly

8 The conditional GO TO verb that can be used to implement the *case* type of program structure in COBOL is the GO TO . . . ______________ verb.

DEPENDING ON

9 The EVALUATE verb provides still another way of implementing the *case* type of program structure in COBOL. It is available in [only the new language revision / both the 1974 standard and the new revision].

only the new language revision

CLASS CONDITIONS

The use of a class condition test makes it possible to determine whether the contents of a data field are numeric, alphabetic, alphabetic lowercase, or alphabetic uppercase. Further, by the use of a combination of such conditionals, we also can determine if the field contains alphanumeric data. The general format for the class condition test is:

```
                           { NUMERIC          }
IF identifier IS [NOT]     { ALPHABETIC       }
                           { ALPHABETIC-LOWER }
                           { ALPHABETIC-UPPER }
```

A data field is numeric if it contains only the digits 0–9, with or without an operational sign. Alphabetic items, on the other hand, consist of the letters A–Z

TABLE 6-1

VALID USES OF THE CLASS CONDITION TEST FOR DIFFERENT KINDS OF FIELDS

FIELD CLASS	VALID TEST
Numeric	NUMERIC, NOT NUMERIC
Alphabetic	ALPHABETIC, NOT ALPHABETIC
Alphanumeric	NUMERIC, NOT NUMERIC, ALPHABETIC, NOT ALPHABETIC

and/or blanks. It is not valid to perform a NUMERIC class test on an alphabetic field or an ALPHABETIC class test on a numeric field. Thus, suppose we have the following DATA DIVISION specifications:

```
AMOUNT    PICTURE 9(4)V99.
NAME      PICTURE A(15).
```

It would be improper to write:

```
IF AMOUNT IS ALPHABETIC . . . or IF NAME IS NUMERIC . . .
```

Instead, the AMOUNT field just given can be tested to determine if the content is in fact NUMERIC or if it is NOT NUMERIC. Similarly, the NAME field can be tested only to determine if the content is ALPHABETIC or NOT ALPHABETIC. A common case of a numeric field not containing numeric data involves reading a field from a record that contains one or more blanks. Specifying the PICTURE with 9s does not guarantee that the field will contain numeric digits. Table 6-1 summarizes the valid uses of the class condition test for different kinds of fields. Note that any of the condition tests may be used with an alphanumeric field.

In general, the class condition test is useful as a check to determine if particular data classes contain the type of data as defined in the DATA DIVISION: numeric, alphabetic, or alphanumeric. The tests for NUMERIC and ALPHABETIC are straightforward, such as:

```
IF AMOUNT IS NUMERIC . . .
IF NAME IS NOT ALPHABETIC . . .
```

In effect, the first statement directly tests the appropriateness of the content in the numeric field called AMOUNT, whereas the second statement tests for inappropriateness of the content in the alphabetic field called NAME. Since an alphanumeric field can have both alphabetic and numeric content, the alphanumeric content can be ascertained indirectly by determining that the content is not entirely numeric and that it is not entirely alphabetic, as follows:

```
IF  FIELD-A IS NOT NUMERIC
    IF  FIELD-A IS NOT ALPHABETIC
        PERFORM ALPHA-NUM-PAR . . .
```

The class condition test cannot be used with numeric items whose USAGE has been declared COMPUTATIONAL. Thus the usage must be explicitly or implicitly DISPLAY. The COMPUTATIONAL and DISPLAY clauses are discussed in Chapter 8: "Additional DATA and PROCEDURE DIVISION Statements."

Review

1 The purpose of a class condition test, is to determine if the actual content of a storage field is ________________, ________________, or ________________.

numeric, alphabetic, alphanumeric

2 Suppose that a field named VENDOR has been defined as an alphabetic field in the DATA DIVISION. If we wish to check for the possibility that numeric data have been entered into this field, we could do so by the statement: IF VENDOR IS ______________.

NOT ALPHABETIC

3 Suppose that a field named ADDRESS has been defined as an alphanumeric field in the DATA DIVISION. If we wish to ascertain that the content of the field is in fact alphanumeric, we can do so by the statement ______________.

IF ADDRESS IS NOT NUMERIC
IF ADDRESS IS NOT ALPHABETIC

USING CONDITIONALS TO CHECK INPUT DATA

The well known GIGO (Garbage In Garbage Out) acronym summarizes effectively the importance of accurate input data. Before data can be processed, they must be checked for correctness and completeness. Errors creep into source data for a variety of reasons, but are due mainly to human error. A good data processing system accepts reasonable error rates in input data as a fact of life and screens the data through so-called edit programs. Such programs use conditional expressions that check input data to determine that they are complete and that they are valid or at least reasonable. Errors that are detected are listed for manual correction and resubmission.

Figure 6-8 presents a simplified program that checks employee records that

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  DATAEDIT.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
*
     SELECT EMPLOYEE-FILE ASSIGN TO  File-name/Device.
     SELECT PRINT-FILE    ASSIGN TO  Printer.
/
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  EMPLOYEE-FILE        LABEL RECORDS ARE OMITTED
                          RECORD CONTAINS 80 CHARACTERS
                          DATA RECORD IS EMPLOYEE-RECORD.
 01  EMPLOYEE-RECORD.
     02 EMPL-NAME                 PIC X(15).
     02 EMPL-NO                   PIC 9(4).
     02 PAY-CODE                  PIC X.
        88  SALARIED              VALUE 'S'.
        88  HOURLY                VALUE 'H'.
     02 PAY-RATE                  PIC 9(4)V99.
     02 FILLER                    PIC X(54).
*
 FD  PRINT-FILE           LABEL RECORDS ARE OMITTED
                          RECORD CONTAINS 132 CHARACTERS
                          DATA RECORD IS PRINT-RECORD.
 01  PRINT-RECORD                 PIC X(132).
```

FIGURE 6-8
LISTING OF THE PROGRAM TO CHECK INPUT DATA.

```
*
WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-INDICATOR  PIC XXX VALUE 'NO '.
     88 END-OF-FILE         VALUE 'YES'.
*
 01  RECORD-PRINT-SWITCH    PIC 9 VALUE ZERO.
     88 RECORD-HAS-BEEN-PRINTED VALUE 1.
*
 01  PAY-LIMITS.
     02 MINIMUM-SALARY      PIC 9(4)V99 VALUE  600.00.
     02 MAXIMUM-SALARY      PIC 9(4)V99 VALUE 6800.00.
     02 MINIMUM-WAGE        PIC 9(4)V99 VALUE    3.50.
     02 MAXIMUM-WAGE        PIC 9(4)V99 VALUE   18.99.
*
 01  ERROR-MESSAGE-RECORD.
     02 FILLER              PIC X(20) VALUE SPACES.
     02 ERROR-MESSAGE       PIC X(60).
 01  INPUT-RECORD-OUT.
     02 FILLER               PIC X(5) VALUE SPACES.
     02 INPUT-RECORD         PIC X(80).
*
 01  HEADING-1.
     02 FILLER               PIC X(30) VALUE SPACES.
     02 FILLER PIC X(24) VALUE 'INPUT DATA ERROR LISTING'.
*
 01  HEADING-2.
     02 FILLER              PIC X(27) VALUE SPACES.
     02 FILLER              PIC X(36) VALUE
      '(RECORD PRECEDES ITS ERROR MESSAGES)'.
/
PROCEDURE DIVISION.
*
PROGRAM-SUMMARY.
*
    OPEN INPUT   EMPLOYEE-FILE
         OUTPUT  PRINT-FILE.
*
    PERFORM READ-RECORD
*
    WRITE PRINT-RECORD FROM HEADING-1 AFTER PAGE
    WRITE PRINT-RECORD FROM HEADING-2 AFTER 1.
*
    PERFORM ERROR-CHECKING  UNTIL END-OF-FILE.
*
    CLOSE EMPLOYEE-FILE,  PRINT-FILE.
*
    STOP RUN.
*
READ-RECORD.
*
    READ EMPLOYEE-FILE RECORD
         AT END MOVE 'YES' TO END-OF-FILE-INDICATOR.

*
ERROR-CHECKING.
*
    IF EMPL-NAME = SPACES
       MOVE 'EMPLOYEE NAME IS MISSING' TO ERROR-MESSAGE
       PERFORM ERROR-PRINT.
*
    IF EMPL-NO NOT NUMERIC
       MOVE 'EMPLOYEE NUMBER CONTAINS NON-NUMERIC CHARACTERS'
           TO ERROR-MESSAGE
       PERFORM ERROR-PRINT.
*
    IF (NOT SALARIED) AND (NOT HOURLY)
       MOVE 'PAY CODE IS NEITHER S NOR H' TO ERROR-MESSAGE
       PERFORM ERROR-PRINT.
```

FIGURE 6-8 *(Continued)*
LISTING OF THE PROGRAM TO CHECK INPUT DATA.

```
*
     IF PAY-RATE NOT NUMERIC
        MOVE 'PAY RATE CONTAINS NON-NUMERIC CHARACTERS'
            TO ERROR-MESSAGE
        PERFORM ERROR-PRINT.
     IF (NOT SALARIED) AND (NOT HOURLY)
        NEXT SENTENCE
     ELSE
        IF SALARIED
            AND (PAY-RATE < MINIMUM-SALARY)
            OR  (PAY-RATE > MAXIMUM-SALARY)
           MOVE 'UNREASONABLE PAY RATE FOR SALARIED EMPLOYEE'
                TO ERROR-MESSAGE
           PERFORM ERROR-PRINT
        ELSE
           IF HOURLY
               AND (PAY-RATE < MINIMUM-WAGE)
               OR  (PAY-RATE > MAXIMUM-WAGE)
              MOVE 'UNREASONABLE PAY RATE FOR HOURLY EMPLOYEE'
                   TO ERROR-MESSAGE
              PERFORM ERROR-PRINT
           ELSE
              NEXT SENTENCE.
*
     MOVE ZERO TO RECORD-PRINT-SWITCH
*
     PERFORM READ-RECORD.
*
 ERROR-PRINT.
*
     IF RECORD-HAS-BEEN-PRINTED
        NEXT SENTENCE
     ELSE
        MOVE 1 TO RECORD-PRINT-SWITCH
        MOVE EMPLOYEE-RECORD TO INPUT-RECORD
        WRITE PRINT-RECORD FROM INPUT-RECORD-OUT AFTER 3.
*
     WRITE PRINT-RECORD FROM ERROR-MESSAGE-RECORD AFTER 1.
```

FIGURE 6-8 *(Continued)*
LISTING OF THE PROGRAM TO CHECK INPUT DATA.

contain payroll data. Sample input and output are shown in Figure 6-9. As the program illustrates, it is a good procedure to list the record that contains the error and to check for all possibilities. If there is one error, the chances are that there may be several errors, and we should identify all of them the first time through. An error message is printed for each error that is detected. It is worth noting that the PAY-RATE field is checked for reasonableness by comparing it to maximum and minimum limits, but such a test is meaningful only if the PAY-CODE is correct (either salaried or hourly).

Review

1 A good data processing system is based on the assumption that there will be [no / some] errors in input data.

some

2 The purpose of a typical edit program is to test for the [accuracy / reasonableness] of input data.

reasonableness

3 For each apparent error detected by use of an edit program, it is good practice to [list only the item in error / list all dedectable errors].

list all detectable errors

```
BROWN,R.K.       12345156035
DAVIS,M.O.        5246H250603
GARCIA,L.A.      3345H000676
HARRISON,P.N     21005000700
MARTIN,A.C.      5123512000
MARTINEZ,P.M.    44335800000
PETERSON,R.A.    6161H0001256
```

```
                              INPUT DATA ERROR LISTING
                          (RECORD PRECEDES ITS ERROR MESSAGES)

BROWN,R.K.       12345156035
                 PAY CODE IS NEITHER S NOR H

DAVIS,M.O.        5246H250603
                 EMPLOYEE NUMBER CONTAINS NON-NUMERIC CHARACTERS
                 PAY CODE IS NEITHER S NOR H
                 PAY RATE CONTAINS NON-NUMERIC CHARACTERS

HARRISON,P.N     21005000700
                 PAY CODE IS NEITHER S NOR H

MARTIN,A.C.      5123512000
                 PAY CODE IS NEITHER S NOR H
                 PAY RATE CONTAINS NON-NUMERIC CHARACTERS

MARTINEZ,P.M.    44335800000
                 PAY CODE IS NEITHER S NOR H

PETERSON,R.A.    6161H0001256
                 UNREASONABLE PAY RATE FOR HOURLY EMPLOYEE
```

FIGURE 6-9

SAMPLE INPUT AND OUTPUT FOR THE PROGRAM TO CHECK INPUT DATA.

SIGN CONDITIONS

The sign condition determines whether or not the algebraic value of an identifier or arithmetic expression is greater than, less than, or equal to, zero. The general format for the sign condition is:

```
IF { identifier            }  IS [NOT]  { POSITIVE }  . . .
   { arithmetic-expression }            { NEGATIVE }
                                        { ZERO     }
```

The subject of the condition must be a numeric field or arithmetic expression. If the value contained in the field is greater than zero, it is POSITIVE; if the value is equal to zero, it is ZERO; and if it is less than zero, it is NEGATIVE.

As was true for the class condition test, the sign condition is used frequently as a check on the appropriateness of data. For example, if an inventory figure cannot be negative by definition, the presence of a negative value in such a field indicates some kind of error. In other circumstances, a zero or negative value in an inventory field might be indicative of an out-of-stock condition, and the test could be used to initiate a reordering procedure.

Review

1 The sign condition can be used to test for three specific types of conditions in regard to the content held in a particular field: whether it is positive, __________, or __________.

negative; zero

2 The subject of the sign condition must be a __________ field.

numeric

CONDITION-NAME CONDITIONS

In Chapter 4, "DATA DIVISION Features," we described the use of condition-names, which are identified by level number 88. For completeness and later reference, we include such conditions here. Let us consider an example:

```
02 EMPLOYEE-CODE        PIC 9.
   88 MALE              VALUE 1.
   88 FEMALE            VALUE 2.
   88 ERROR-SEX-CODE    VALUES ARE ZERO, 3 THRU 9.
```

This example illustrates a typical programming error. One would reason that since EMPLOYEE-CODE is a one-digit numeric field and since MALE and FEMALE account for the 1 and 2 values, ERROR-SEX-CODE correctly covers *all* other possibilities. However, this is not true. The PIC 9 specification does not mean that the field must contain numeric data. As we explained in Chapter 5 in conjunction with discussion of the MOVE statement, a PIC 9 field may contain *nonnumeric* data. Thus, a correct procedure for checking for errors would be to check both whether EMPLOYEE-CODE is not NUMERIC and if it is numeric, then to check whether it is zero and 3 through 9. Still, we would like to write a condition-name that covers all possibilities of an error code more simply. Here is a rewritten specification:

```
02 EMPLOYEE-CODE        PIC X.
   88 MALE              VALUE '1'.
   88 FEMALE            VALUE '2'.
   88 ERROR-SEX-CODE    VALUES ARE LOW VALUES THRU ZERO
                                   '3' THRU HIGH-VALUES.
```

In the above program segment we recognize the fact that EMPLOYEE-CODE is not a *numeric* field in the quantitative sense. The code values are simply characters that identify each employee; they are not measures of quantity. Thus a code of 2 is not a larger value than a code of 1; they are simply different values. Recognizing the nominal nature of the code, it makes more sense to use PIC X rather than PIC 9, and then the 88-level condition-name can be used to capture all error cases. Specifically, the LOW-VALUES figurative constant refers to the smallest character in the computer and HIGH-VALUES refers to the highest value (see Figure 1-4). This approach would "sense" *any* incorrect value in the field.

As a last point related to the example above, we should point to the use of nonnumeric literals such as '1', '2', and '3'. For example, the MALE condition-name refers to EMPLOYEE-CODE having the character 1 as its content. The PIC X specification defines the field as alphanumeric and we should stay consistent with that definition when using literals. It would not be correct to define VALUE 1 (rather than VALUE '1') in conjunction with the PIC X specification.

Review

1 Condition-names are identified by the level number ________________ in the DATA DIVISION.

88

2 Given that a numeric code is used to identify categories rather than quantities in condition-name descriptions, then the preferred PICTURE for such a one-digit code is [PIC 9 / PIC X].

PIC X

3 When PIC X specification is used for a data item used for condition-names, then a condition represented by the numeric code 3 correctly has the defined value [VALUE 3 / VALUE '3'].

VALUE '3'

THE SET VERB FOR CONDITION-NAMES

The revised language provides for a convenient way to move data into fields for which condition-names have been defined. The 88-level condition-name feature of COBOL is very meaningful for conducting tests as to whether a condition is true. But in the 1974 version of COBOL there is something lacking with respect to moving values that correspond to defined condition-names. Let us consider an example.

```
01  TOTAL-CREDIT-HOURS     PIC 999.
01  STUDENT-CLASS-CODE     PIC X.
    88  FRESHMAN VALUE     '1'.
    88  SOPHOMORE VALUE    '2'.
    88  JUNIOR VALUE       '3'.
    88  SENIOR VALUE       '4'.
IF TOTAL-CREDIT-HOURS < 33
      MOVE '1' TO STUDENT-CLASS-CODE
ELSE IF TOTAL-CREDIT-HOURS < 65
        MOVE '2' TO STUDENT-CLASS-CODE
     ELSE IF TOTAL-CREDIT-HOURS < 97
             MOVE '3' TO STUDENT-CLASS-CODE
          ELSE MOVE '4' TO STUDENT-CLASS-CODE.
```

In the above program segment, statements like MOVE '1' TO STUDENT-CLASS-CODE do not take advantage of the 88 FRESHMAN condition-name specification. But consider the following program segment:

```
IF TOTAL-CREDITS < 3
      SET FRESHMAN TO TRUE
ELSE IF TOTAL-CREDITS < 65
          SET SOPHOMORE TO TRUE
     ELSE IF TOTAL-CREDITS < 97
              SET JUNIOR TO TRUE
          ELSE SET SENIOR TO TRUE.
```

The SET verb has been used to take advantage of the convenient condition-names provided by the 88-level items.

The general format of the SET verb for this purpose is given below. There are also other uses of the SET verb and they are presented in Chapter 12, on table-handling features.

```
SET {condition-name-1} . . . TO TRUE
```

In this format the SET verb specifies that the value specified in the VALUE clause is to be moved to the field involved. If a range of values is specified, then the first value is moved, as in the following example.

```
01 STUDENT-CLASS-CODE PIC X.
   88 UNDERCLASSMAN   VALUES ARE '1' THRU '2'.
   88 UPPERCLASSMAN   VALUES ARE '3' THRU '4'.
```

Use of SET UNDERCLASSMAN TO TRUE then is equivalent to saying MOVE '1' TO STUDENT-CLASS-CODE.

If multiple condition-names are specified in the same SET statement, the effect is the same as using multiple SET statements. This is illustrated in the following example.

```
01 CREDIT-RATING PIC XX.
   88 DOUBLE-A   VALUE 'AA'.
   88 GOOD       VALUES ARE 'A' THRU 'BB'.
01 ACCOUNT-CLASS PIC X.
   88 INSTITUTIONAL  VALUE 'I'.
   88 PERSONAL       VALUE 'P'.
   .
   SET DOUBLE-A PERSONAL TO TRUE.
```

The above SET statement is equivalent to:

```
SET DOUBLE-A  TO TRUE
SET PERSONAL  TO TRUE.
```

The latter style is the recommended approach, because it makes it clear that two condition names are set to true. Of course, in either form it would not be logical to use two condition-names from the *same* field, such as:

```
SET DOUBLE-A TO TRUE
SET GOOD     TO TRUE.
```

The 88 condition-names included in a given field are mutually exclusive categories, and only one of them can apply.

Review

1 The verb that can be used to move values that correspond to defined condition-names by reference to the condition-name itself is the ________ verb.

SET

2 The SET verb is available in the [1974 version of COBOL / revised language only].

revised language only

3 Given that the DATA DIVISION specification is 88 MALE VALUE '1', then the correct SET verb statement for an individual that is a male is: SET MALE TO ________________.

TRUE

COMPLEX CONDITIONS

It is possible to combine the simple (individual) conditionals we have described into complex conditionals by the use of the logical operators OR, AND, and NOT. OR means either or both, and AND means both. Thus, consider the following statement:

```
IF BALANCE IS NEGATIVE AND DAYS-OVERDUE > 10
    PERFORM PAR-A.
```

The instruction indicates that the program should execute PAR-A when both the balance is negative and the number of overdue days exceeds 10.

On the other hand, consider the following statement:

```
IF INPUT-DATA IS NOT NUMERIC OR NAME-IS-MISSING
    MOVE 'CANT PROCESS, INCORRECT DATA' TO ERR-MESSAGE.
```

The program will move the indicated message to ERR-MESSAGE if either the input data are not numeric (perhaps because of a keying error) or the condition-name condition defined as NAME-IS-MISSING in the DATA DIVISION holds.

There is a rather complex set of rules associated with the writing and evaluation of complex conditionals. From the standpoint of the scope and orientation of this book, however, we shall limit our attention to the use of parentheses to clarify the meaning. For example, we can write:

```
IF (AGE IS GREATER THAN 28) OR ((EXPERIENCE = 4)
    AND (EDUCATION IS GREATER THAN HS)) . . .
```

This condition holds either if age is greater than 28 or if both experience = 4 and education is greater than high school.

As another example, consider the following:

```
IF ((KODE = 2) OR (KODE = 3)) AND (BALANCE-CODE = 1)
    MOVE SPACES TO ERROR-MESSAGE
    PERFORM OLD-ITEM-2.
```

In this example, the condition is true if BALANCE-CODE is equal to 1 and KODE is either equal to 2 *or* equal to 3.

Thus, by the use of complex conditionals, we can write conditional tests that otherwise would require very long expressions consisting of several nested IF statements. Complex conditionals are useful particularly for selecting data from a file, since we can designate selective retrieval of records according to the presence or absence of complex conditions.

Review

1 In contrast to the use of simple conditionals, a combination of tests can be included in one statement by the use of ________________ conditionals.

complex

2 The use of a complex conditional requires the use of one of the logical operators: ________________, ________________, or ________________.

OR; NOT; AND

3 When the logical operator OR is used in a complex conditional test, the *presence of [either / both / either or both] of the conditional states constitutes* a YES condition.

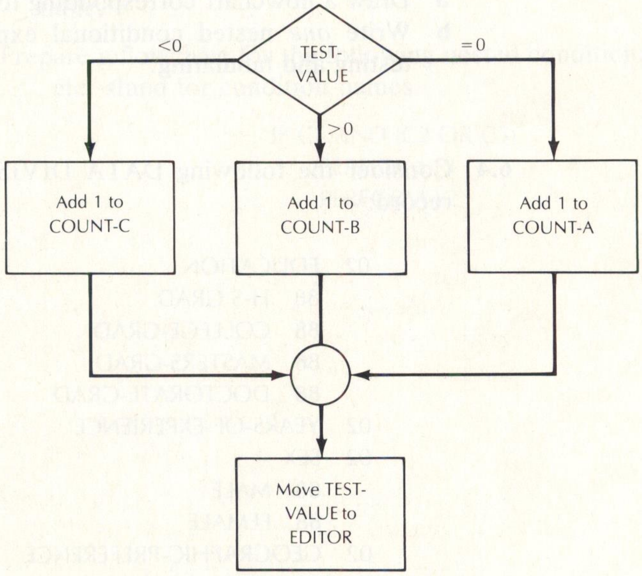
either or both

4 When the logical operator AND is used in a complex conditional test, the presence of [either / both / either or both] of the conditional states constitutes a YES condition.

both

EXERCISES

6.1 Write PROCEDURE DIVISION statements to implement the logic included in the following flowchart.



6.2 An input field has been defined as:

```
03  IN-FIELD     PICTURE X(10).
```

Two other fields in WORKING-STORAGE have been defined as:

```
01  AMOUNT       PICTURE 9(10).
01  NAME         PICTURE A(10).
```

We wish to test the content of IN-FIELD and, if it contains a number, to store it in AMOUNT; if it contains a name, to store it in NAME; and if a mixture of characters, to go to a paragraph called ERRORS.

- **a** Write PROCEDURE DIVISION statements to accomplish this task.
- **b** Suppose that, if the IN-FIELD contains a number, it is actually in dollars and cents. What would you do to make the number available in dollars and cents instead of as an integer? (*Hint:* Be careful!)

6.3 Using the 88 level number indicator in the DATA DIVISION and suitable condition-name clauses, the following obvious identifiers have been defined:

MALE
FEMALE
SINGLE-M

```
MARRIED-M
DIVORCED-M
WIDOWED-M
SINGLE-F
MARRIED-F
DIVORCED-F
WIDOWED-F
```

Assume we want to tabulate the number of individuals falling in the last eight classes, as, for example, the number of single males (SINGLE-M). We thus want to test the field containing the identifying code and ADD 1 TO the corresponding counter. Assume the following fields are to be used as counters: SM, MM, DM, WM, SF, MF, DF, WF (where SM stands for single males, etc.).

a Draw a flowchart corresponding to your program logic.
b Write *one* nested conditional expression to accomplish the required testing and tabulating.

6.4 Consider the following DATA DIVISION entries relating to a personnel record:

```
02  EDUCATION                                PICTURE 99.
    88  H-S GRAD               VALUE 12.
    88  COLLEGE-GRAD           VALUE 16.
    88  MASTERS-GRAD           VALUE 17.
    88  DOCTORATE-GRAD         VALUE 20.
02  YEARS-OF-EXPERIENCE                      PICTURE 99.
02  SEX                                      PICTURE 9.
    88  MALE                   VALUE 1.
    88  FEMALE                 VALUE 2.
02  GEOGRAPHIC-PREFERENCE                    PICTURE 9.
    88  EAST                   VALUE 1.
    88  MIDWEST                VALUE 2.
    88  WEST                   VALUE 3.
    88  SOUTH                  VALUE 4.
    88  WILLING-TO-TRAVEL      VALUE 5.
```

Suppose that we want to find individuals who fulfill one of these three requirements:

a Five years of experience, male, high school graduate, willing to travel
b Male, one year of experience, master's degree, preferring the West or South
c Three years of experience, female, doctorate, preferring the East

Write *one* compound conditional sentence by which we can check whether a record in question fulfills the first, second, or third of these requirements. If one of these requirements is met, we WRITE PRINTLINE FROM NAME. If the requirement is not met, we execute PAR-A.

Instead of the above, use the EVALUATE verb to implement the program logic.

6.5 Consider the following table of conditions:

QUANTITY	PRICE	RATING	DISCOUNT
>100	>10	<2	0.05
−100	>10	≥2	0.10
−100	≤10	<2	0.15
>100	≤10	≥2	0.20
≤100	{ < / = 10 / > }	{ < / = 2 / > }	0.25

a Write instructions—using nested IF—to MOVE to DISCOUNT the value shown depending on the conditions. Do not use END-IF.

b Use the END-IF scope terminator to do the task.

c Use the EVALUATE verb to do the task.

d Prepare a flowchart corresponding to the specifications in the above table.

6.6 Prepare a flowchart for the following nested conditional statement. C1, C2, . . . etc. stand for condition-names.

```
IF C1 AND (C2 OR C3)
    PERFORM F1
    PERFORM F2
```

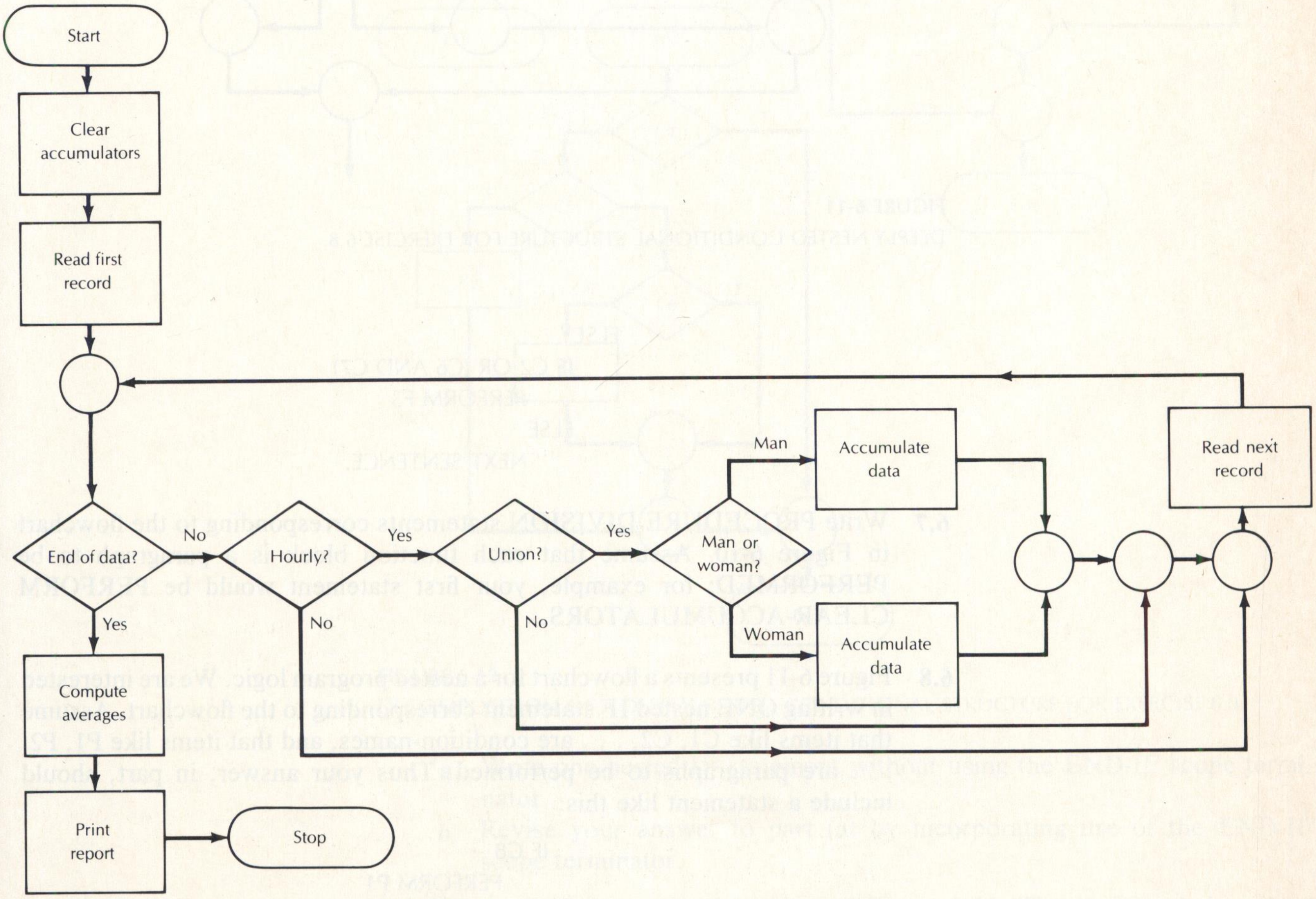


FIGURE 6-10
FLOWCHART FOR EXERCISE 6.7.

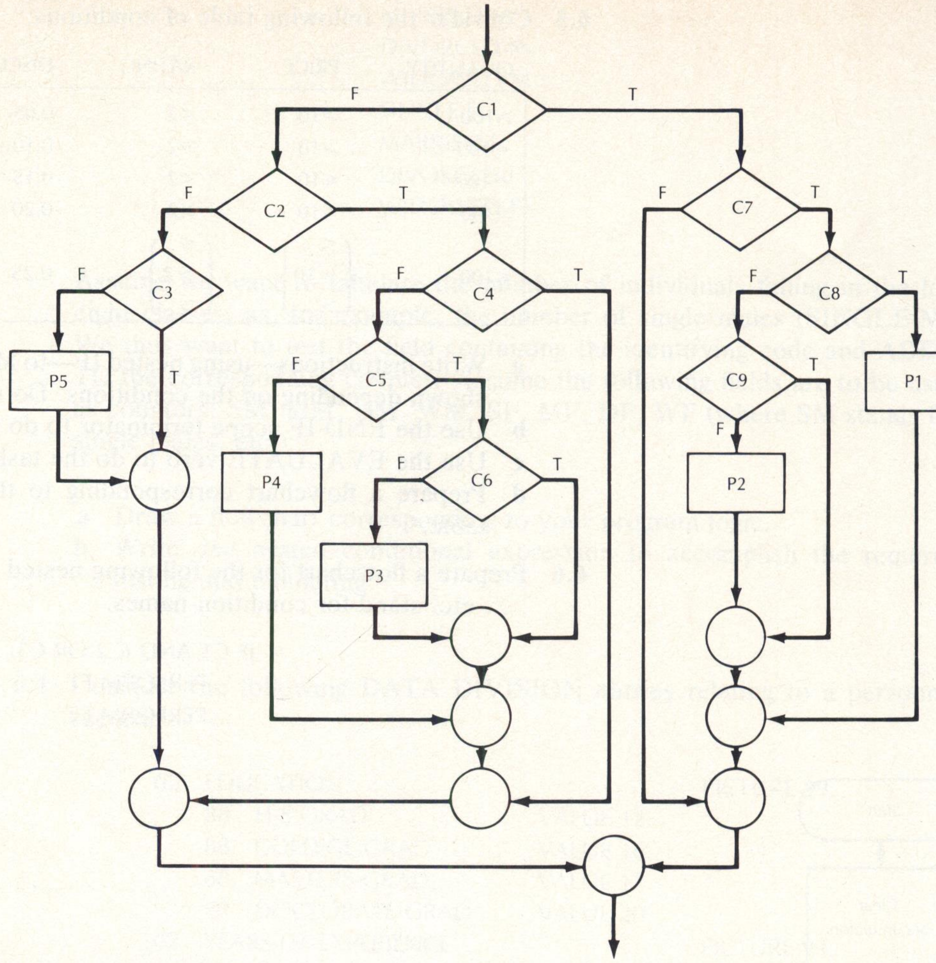


FIGURE 6-11
DEEPLY NESTED CONDITIONAL STRUCTURE FOR EXERCISE 6.8.

```
ELSE
      IF C2 OR (C6 AND C7)
            PERFORM F3
      ELSE
            NEXT SENTENCE.
```

6.7 Write PROCEDURE DIVISION statements corresponding to the flowchart in Figure 6-10. Assume that each function block is a paragraph to be PERFORMED; for example, your first statement would be PERFORM CLEAR-ACCUMULATORS.

6.8 Figure 6-11 presents a flowchart for a nested program logic. We are interested in writing ONE nested IF statement corresponding to the flowchart. Assume that items like C1, C2, . . . are condition-names, and that items like P1, P2, . . . are paragraphs to be performed. Thus your answer, in part, should include a statement like this:

```
IF C8
      PERFORM P1
          ·
          ·
          · (etc.)
```

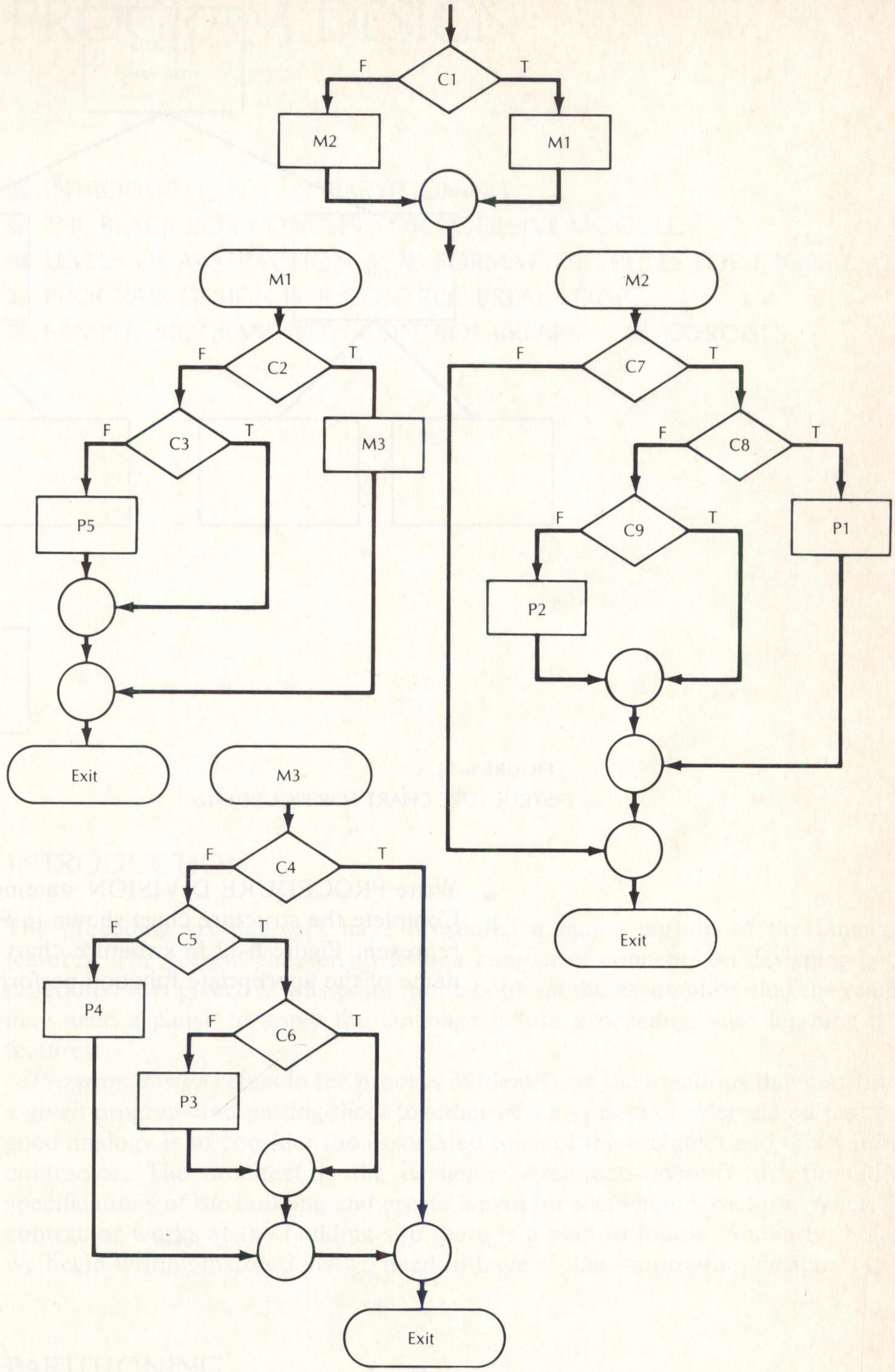


FIGURE 6-12
DECOMPOSITION OF A DEEPLY NESTED CONDITIONAL STRUCTURE FOR EXERCISE 6.9.

a Write one nested IF statement without using the END-IF scope terminator.

b Revise your answer to part (a) by incorporating use of the END-IF scope terminator.

6.9 Figure 6-12 is a revised version of Figure 6-11. The revision is based on partitioning the one deeply nested structure in Figure 6-11 into several, smaller modules.

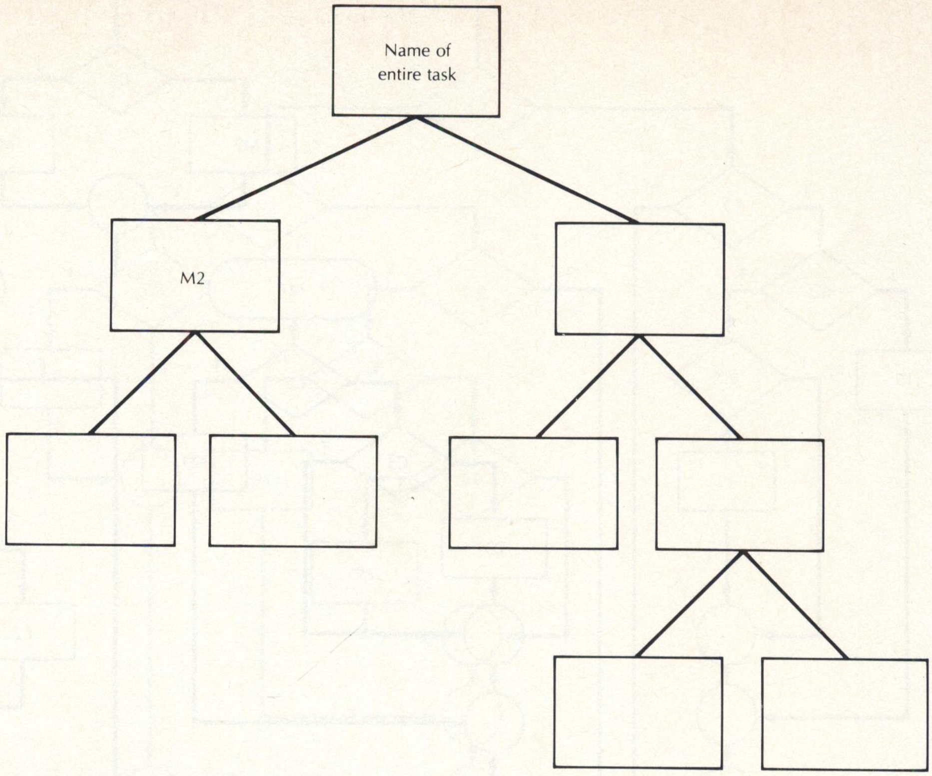


FIGURE 6-13
STRUCTURE CHART FOR FIGURE 6-12.

a Write PROCEDURE DIVISION statements to represent Figure 6-12.
b Complete the structure chart shown in Figure 6-13 which is designed to represent Figure 6-12 in structure chart form. In each block, write the name of the appropriate function performed by the program.

7 PROGRAM DESIGN

INTRODUCTION

The preceding six chapters have presented a major portion of the language features. The current chapter presents a number of concepts on designing good programs. It is placed at this point in the book on the assumption that the reader may need a pause to apply the language before proceeding with learning new features.

Program design refers to the process of identifying the functions that constitute a given program and putting them together into a system of interrelated parts. A good analogy is to consider the associated roles of the architect and the building contractor. The architect is the designer. Architects identify the functional specifications of the building and create a plan for the whole structure. When the contractor works at the building site there is a plan to follow. Similarly, before we begin writing instructions we need to have a plan—a program design.

PARTITIONING

A fundamental concept of program design is that of *partitioning,* which refers to the process of subdividing a large programming task into smaller parts or functions.

Partitioning is a common phenomenon in human activities. One form of partitioning in organizations is based on the division of labor, or functional specialization. For example, an automobile manufacturing plant includes departmental units which may be further subdivided according to specific functions. A Painting Department, for instance, could include such separate functions as cleaning, spraying, baking, inspecting, and the like. Similarly, an Electronic Data Processing Department could include such separate functions as programming,

systems analysis, data entry, and input-output control. The common occurrence of partitioning in a variety of situations reflects the physical and mental limits of human beings. A given person can only do so much and attend to so much at a given time. Therefore, we find it not only beneficial, but also necessary, to partition large and complex tasks into smaller and more specialized tasks.

A computer programming task generally is complex enough to make partitioning desirable. From the standpoint of the individual programmer, the partitioning of the overall task allows the programmer to concentrate on particular program functions. From the standpoint of the organization, partitioning makes it possible to complete complex programming tasks in a shorter time by having a team of programmers working simultaneously on different specific tasks that constitute the overall program.

In the context of computer programming, a widely used term associated with partitioning is *modularity*. A program module is a well-defined program segment. Modular programming has been recognized as a desirable practice for many years. In practice, all programs include some degree of modularity by necessity: no programmer can write a monolithic program that is not partitioned into some kinds of parts, or modules. Thus, it is not just presence of modularity that is important. Rather, we need to develop an understanding of how to design programs whose modules are so constructed as to lead to good programs.

To be useful, a module should not only be a program segment, but also a *well-defined* program segment. More specifically, a module should be a named program segment that carries out a specific program function. In the context of COBOL programming, a module eventually is represented in one of four forms in the program:

1 As a single paragraph
2 As a series of two or more consecutive paragraphs which are the object of a PERFORM PA THRU PZ, where PA and PZ stand for the first and last paragraphs
3 As a single section
4 As a subprogram

Review

1 The process of subdividing a large programming task into smaller, more specific tasks is called ________________.

partitioning

2 In terms of human endeavors, partitioning is a [long-standing / recently developed] concept.

long-standing

3 "A named program segment that carries out a specific program function" is a definition of a program ________________.

module

4 Program modularity can be described as being effective when it leads to the development of ________________ programs.

good

THE BLACK BOX CONCEPT FOR COHESIVE MODULES

In order to achieve good program design, a program module should have the property of being a black box. The *black box concept* means that a module should

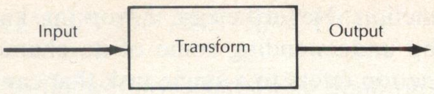


FIGURE 7-1
THE BLACK BOX CONCEPT.

be capable of being described in terms of the *function* that it performs, rather than the *procedure* by which it performs the function. In other words, we should be able to describe *what* the module is designed to do rather than *how* it is done. The black box concept originated in engineering applications, and in that context it is conventional to speak of the input, the transform, and the output of a black box. Figure 7-1 illustrates this viewpoint schematically.

In the context of programming, the *input* to a black box consists of data supplied to the module and the *output* consists of data generated by the module. The *transform* consists of the combined effect of the series of statements that comprise the module. When the module is a single paragraph, a series of paragraphs, or even a section, the input and output data are not directly obvious in COBOL programs. The reason is that all of the data are defined in the one DATA DIVISION, and thus the input and output are specified indirectly. For instance, consider the following program segment:

```
        .
        .
        .
        PERFORM READ-TRANS-REC
        MOVE 'NO' TO FILE-SEQ-FLAG
        PERFORM CHECK-FILE-SEQ
        IF FILE-SEQ-FLAG = 'YES' . . .
        .
        .
        .
        CHECK-FILE-SEQ.
            IF CUST-NO OF TRANS-REC < PREV-CUST-NO
                MOVE 'YES' TO FILE-SEQ-FLAG
            ELSE
                MOVE CUST-NO OF TRANS-REC TO PREV-CUST-NO.
```

The input to the CHECK-FILE-SEQ module consists of FILE-SEQ-FLAG, CUST-NO, and PREV-CUST-NO, while the output consists of the same three data items but with some changed values. Thus, even though the form of the program does not identify the input and output directly, a paragraph as a module does comply with the general black box concept.

In the case where a module is a program subroutine, the arguments of the CALL statement identify explicitly the input and output of the module, as explained in Chapter 15, "Subprograms and Nested Programs." Therefore, regardless of the way that a module is represented in a program, from the design standpoint every module is a black box that can be described in terms of the input, the function performed (transformation), and the output.

In addition to each module representing a black box, a well-designed program should consist of modules that have as much cohesion as possible. A module is *cohesive* if it performs only one function and all program statements in the module are directly related to that function only. Thus, a well-partitioned program ideally should consist of specific functions each of which is represented by a cohesive module in the program.

From the practical standpoint, the question that arises is: What is a cohesive function? Unfortunately, there is no specific and comprehensive definition of a

cohesive function. Nevertheless, a working knowledge of this concept can be developed by understanding some of its characteristics. Generally speaking, a cohesive function refers to a single task that can be described in just a few words of language. An example is: "Compute this week's F.I.C.A. tax for the employee." But what about the function: "Process the payroll for this week"? The latter function is too broad, and should be partitioned into several more specific tasks or functions. At the other extreme, however, the printing of a line of information or the addition of two numbers are tasks that are too specific to be represented as distinct program modules. Thus both experience and judgment serve as the basis for defining cohesive program modules. Still, the following two guidelines can be useful:

- A cohesive module, when coded in COBOL, should not exceed a page of program statements (about 50 lines). However, the module may be represented by only 1 or 2 lines of program code. Typically, a module will be in the range of 5–20 lines of code.
- The module should be capable of being described in ordinary language *without* the use of multiple verbs, the conjunction "and," or use of a series of words such as "first," "then," and the like.

The above guidelines rely on indirect indicators that can alert the programmer to the existence of noncohesive functions.

Review

1 The fact that one should be able to describe a module in terms of the *function* performed, rather than the *procedure* followed, is reflective of the ______________ concept.

black box

2 The three principal elements, or ingredients, associated with the black box concept are the ______________, ______________, and ______________ associated with the black box.

input; transform; output

3 In the context of programming, the input and output associated with a black box are both forms of ______________.

data

4 In COBOL programs, the input and output data are most obvious when the program module represents a [paragraph in the program / program subroutine].

program subroutine

5 A module that performs only one function, and whose program statements are all directed toward that function, is described as being ______________.

cohesive

6 Generally speaking, when a cohesive module is coded in COBOL the program statements should not exceed about ______________ lines.

50

7 Generally speaking, a cohesive module [may / should not] include a series of words such as "first," "then," and "next."

should not

LEVELS OF ABSTRACTION

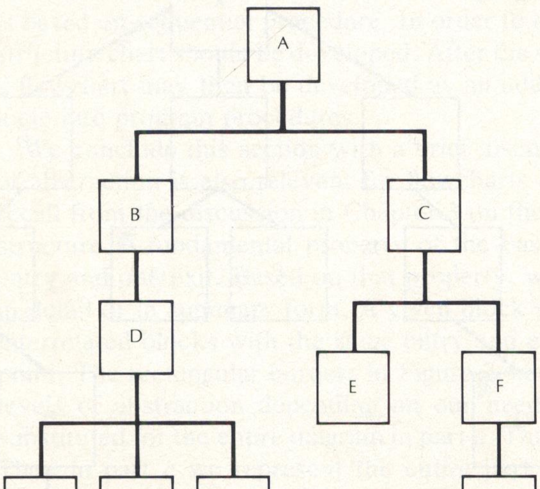
So far we have stated that good programs consist of cohesive modules that can be viewed as black boxes. A black box is a means of representing a program task, a *procedure,* in an abstract way. When we concentrate on the *what* rather than the *how* aspect of a task we introduce the idea of *abstraction,* which is concerned with a summary representation, free of detail.

We stated earlier that an important consideration in program design is the *partitioning* of a whole task into smaller, interrelated modules. In order to achieve effective partitioning of a task, we must be able to look at the parts in abstract form rather than in detail. If a program consists of one hundred modules and each module averages 10 lines of code, it is impossible to comprehend the whole program at once. This is where the black box concept comes in handy. We describe each module as a black box, thus abstracting from its detail. So, for the above example, we would need to consider the one hundred modules rather than the 1,000 lines of code as such. While this is an improvement, we might still find it hard to comprehend the 100 modules viewed all at one time.

The way we manage the large number of modules in program design is to *organize* them in a systematic fashion. Just like a large college that is organized into departments, fields of specialization, and individual faculty, so we organize program modules into a *hierarchical* structure. *Structure charts* are the best way of depicting the hierarchical structure of programs. Figure 7-2 presents two sample hierarchical structure charts, one showing rectangular connecting lines, the second one showing angular connections. The type of connecting lines that are used is simply a matter of preference and has nothing to do with the meaning of a structure chart.

The critical characteristics of a hierarchical structure are:

1. There is only one top module, and it represents the entire program. This module is *superior* or *superordinate* to all other modules.
2. Any given module may have *subordinate* modules. These are shown below their immediate superior and are connected with lines to show this superior-subordinate relationship. In Figure 7-2, for example, D is a module that is subordinate to B and superior to modules G, H, and I.



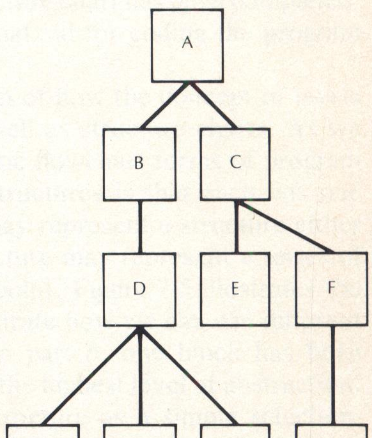


FIGURE 7-2
RECTANGULAR AND ANGULAR STRUCTURE CHARTS.

3 Any given module, except the top one, has one, and only one superior in the structure chart. Yet, there are situations in which the same module may need to be subordinate to more than one superior. In such cases we draw the subordinate module as if it were a unique subordinate to each respective superior in the structure chart, but we shade the upper-right corner to indicate the fact that it is a duplicate representation. Module Q is such a module in Figure 7-3. Notice that we have included the Q block four times in the structure chart, rather than the alternative of presenting it once and showing four connecting lines leading to the module. By this method of duplicate representation, the requirement that a module should have only one superior shown in the structure chart is satisfied.

In all cases, a superior module is superior by the fact that it issues a PERFORM to execute the given subordinate. So in operational terms, a module A is superior to another module B when A contains a "PERFORM B" statement. The PERFORM itself is a means of using abstract references. When a PERFORM is written, it specifies a black box approach; it says what to do, without including the specific procedure of how to do the task. Therefore, a structure chart is an abstract representation of a program because it represents a summary of the functions performed by the program.

Furthermore, a given module is a summary representation of all of its subordinates. Module C in Figure 7-3 is superior to F and Q, which means it contains "PERFORM F" and "PERFORM Q" statements in it. So when execution of C has been completed, F and Q have also been executed. Similarly, when module A in Figure 7-3 has been executed, the entire program has been executed. Thus, a given module provides a high-level, or abstract, representation of its subordinates. The use of abstraction and the omission of detail is our best weapon against overwhelming complexity. A structure chart representing a program of 100 modules need not be overwhelming. The hierarchical structure allows us to focus on parts of the total without losing sight of the whole. When we are viewing one module in order to find out "how it fits in the whole picture,"

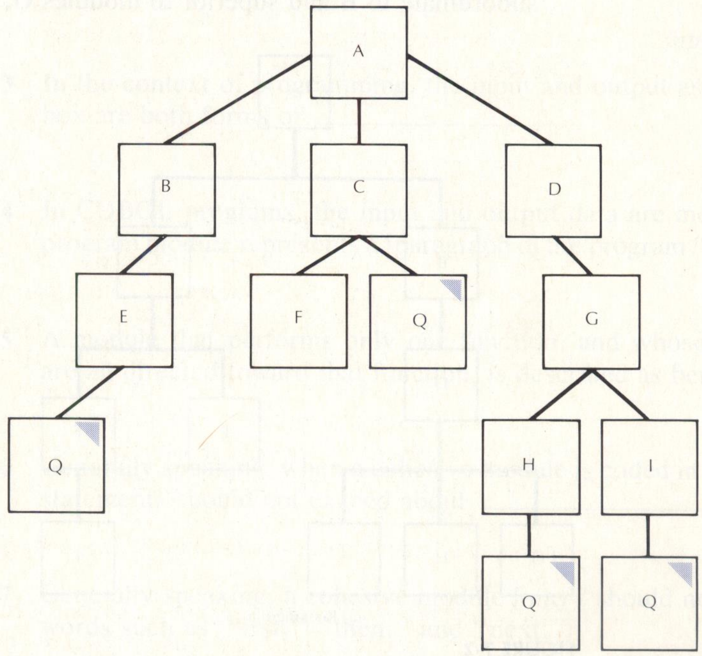


FIGURE 7-3
STRUCTURE CHART ILLUSTRATING A COMMON FUNCTION (Q).

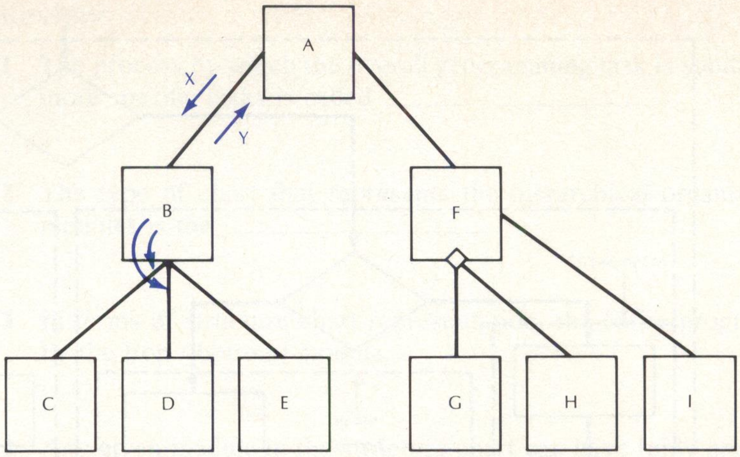


FIGURE 7-4
STRUCTURE CHART SHOWING DATA, LOOPS, AND DECISION POINT.

we simply look up to its superior, and if need be, in turn, to the superior of its superior, and so on.

In addition to describing the hierarchical relationships, a structure chart can also be enhanced to include:

Loops (repetitive execution of subordinate modules)
Decision points (conditional execution of subordinate modules)
Data (input to and output from modules)

Figure 7-4 is a structure chart that includes these additional features. Module A passes X as input data to module B, which returns output data Y to module A. Then, modules C and D are executed repetitively, as indicated by the curved arrows below B; the inner loop involves module C while the outer loop involves both C and D. Finally, module F executes modules G and H conditionally, as indicated by the diamond symbol at F.

It may be useful at this point to contrast a structure chart with a flowchart. A structure chart is based on hierarchical program structure, whereas a flowchart is based on sequential procedure. In order to design a good program, first a good structure chart should be developed. After the structure chart has been completed, a flowchart may then be developed as an additional aid for coding the program logic into program procedures.

We conclude this section with a brief discussion of how the concept of levels of abstraction is also relevant for flowcharts as well as structure charts. As we recall from the discussion in Chapter 3 on the basic flowchart forms of program structure, a fundamental property of the basic structures is that each has one entry and one exit. Based on that property, we may represent a structure either in detail or in summary form. A given block structure may represent a series of interrelated blocks with the same entry and exit point. Figure 7-5 illustrates the point. The rectangular borders in Figure 7-5*a* illustrate how we can use different levels of abstraction depending on our needs. In part *b,* one block has been substituted for the entire diagram in part *a*. This is the highest level of abstraction. Then in part *c* we represent the entire part *a* structure as a simple selection. Finally, part *d* presents more detail by demonstrating the nested selection structure of part *a*.

The one-entry, one-exit property can be used as a powerful tool both when developing a new program and when trying to review or comprehend an existing one. We can deal with minute details of code or we can focus on higher levels of abstraction, as needed.

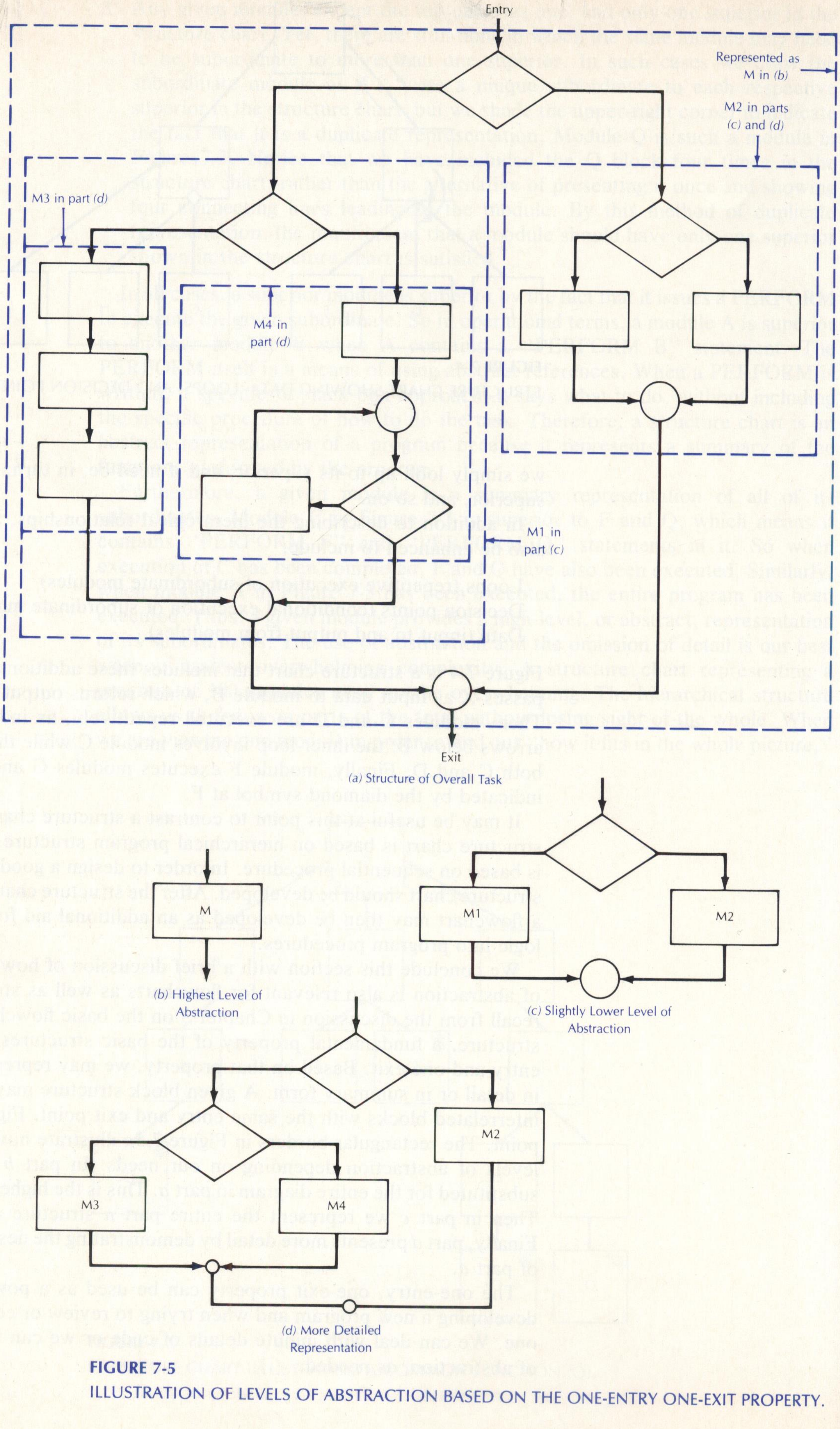


FIGURE 7-5

ILLUSTRATION OF LEVELS OF ABSTRACTION BASED ON THE ONE-ENTRY ONE-EXIT PROPERTY.

Review

1 The process by which the overall programming task is subdivided into several more specific tasks is called ______________.

partitioning

2 The type of chart that represents the hierarchical organization of program modules is the ______________.

structure chart

3 In terms of structure chart representation, the entire program is represented by the [top / bottom] module.

top

4 Any given module in the structure chart can have [only one / any number of] subordinate module(s).

any number of

5 Except for the top module, which has no superior module, any module in the structure chart representation has [only one / any number of] superior module(s).

only one

6 When a given module is in fact subordinate to more than one superior in terms of program logic, that module is ______________ in the structure chart.

repeated (and shaded)

7 If module C is the superior of module D in a structure chart, this indicates that module C includes in it the PROCEDURE DIVISION command, ______________, in abstract form.

PERFORM D

8 As contrasted to the structure chart, which is based on hierarchical program structure, the type of chart that is based on sequential procedure is the ______________.

flowchart

9 A fundamental property of the basic structures is the existence of [only one / several] entry point(s) and [only one / several] exit point(s).

only one; only one

FORMATTING RULES FOR PROGRAMS

Form and substance are highly interrelated. A number of the benefits associated with structured programming derive from use of certain form. Proper substance must be there as a prerequisite to good programming, but proper substance cast in obscure form is hardly worthwhile. In this section we provide some basic guidelines for writing structured programs in readable form. These guidelines are not exhaustive nor are they imperative, but they represent collectively the consensus of good formatting rules practiced in the field.

1 *In the source program use physical spacing to enhance visibility and to denote logical groupings.*

- Use asterisks in column 7 of the COBOL coding form to separate major items. For instance, precede each new 01 item in the WORKING-

STORAGE with an asterisk. Similarly, in the PROCEDURE DIVISION, separate logical groups by a comment line. If there is a series of MOVE statements and then a WRITE, put a blank comment line before the WRITE to block together the MOVEs that perform a common function.

- Use the stroke (/) in column 7 to start a new page. For instance, always begin the DATA DIVISION on a new page and the PROCEDURE DIVISION on a new page. Then within each of these divisions, start a new page when a major logical unit begins. If you have, for example, three 01 heading descriptions and they require about 20 lines of codes, it is much better to start a new page and to precede the program code with a comment such as:

  THE FOLLOWING THREE ITEMS DESCRIBE THE PAGE HEADERS OF THE SALES REPORT.

  Then the reader can quickly grasp the common element in the items on the page, and can choose either to give attention to the page or to bypass it.

- In the PROCEDURE DIVISION it is a good practice to list a PROGRAM SUMMARY type of module on the first page and, if there is enough space, also all its immediate subordinates. If there is not enough space for all the immediate subordinates, then start a new page for each first level module, and, similarly, place all its immediate subordinates on one page.

2 *Use vertical alignment of similar items to convey similarity of function.*

- Align all PICTURE and VALUE clauses on the same column when possible, especially with respect to each 01-level item. For example:

```
01  A.
    02 B    PIC  9(6).
    02 C    PIC  X(2)     VALUE 'ABC'.
    02 D    PIC  99V9.
    02 E    PIC  99V9.
    02 F    PIC  X(3)              VALUE 'XYZ'.
    02 G    PIC  X(29)             VALUE
            'LONG LITERAL ON SEPARATE LINE'.
```

- Align similar verbs and their operands in the PROCEDURE DIVISION. For instance:

```
OPEN   INPUT   FILE-A
               FILE-B
       OUTPUT  FILE-C
MOVE AMOUNT TO ED-AMOUNT
MOVE RATE TO FACTOR
MOVE PREVIOUS-BAL TO TEST-VALUE.
```

- Indent subordinate clauses under the main clause.

  In the DATA DIVISION, for instance, 88 items should be indented as follows:

```
02  TRANS-CODE      PIC  X.
    88  CHANGE-RATE      VALUE '1'.
    88  CHANGE-BAL       VALUE '2'.
    88  ERROR-CODE       VALUE LOW-VALUES THRU ZERO
                               3 THRU HIGH-VALUES.
```

  In the PROCEDURE DIVISION we can use indentation as follows:

```
READ  SOURCE-FILE RECORD
      AT END . . .
```

```
WRITE  DISK-FILE RECORD
       INVALID KEY . . .
PERFORM PROCESS-RATE-CHANGE
          UNTIL AMOUNT-OWED > MAX-LIMIT
                OR ERROR-CODE
MOVE LAST-NAME OF TRANSACTION-FILE-REC
   TO LAST-NAME OF REPORT-REC
CALL 'SUB' USING NO-OF-TRANS
                 POLICY-NO
                 PREMIUM RATE.
```

These illustrations should serve to suggest the various possibilities. There is no reason why one should commit to memory any detailed rules of indentation. Instead, one should develop a general practice and a state of mind to indent to advantage. Within a given organization there may be "standard" indentation rules developed, but petty adherence to any strict set of rules may defeat the main purpose of indentation, which is to facilitate writing readable programs.

3 *Use group labels to convey common functions.*

- In the DATA DIVISION, it is desirable to group similar items under the same group item. For instance, if there are four totals accumulated in a report-generating program, it is preferable to write them under one group name:

```
01  REPORT-TOTALS.
    02  PRODUCT TOTAL
    02  SALESPERSON-TOTAL
    02  DEPARTMENT-TOTAL
    02  GRAND-TOTAL
```

- In the PROCEDURE DIVISION, group labels can be written in many ways. As one approach, a comment line preceding a group of statements may in effect be a group label that explains the common function:

```
*TEST WHETHER GROSS-PAY FALLS WITHIN REASONABLE LIMITS
*
     IF SALARIED-EMPL
        IF GROSS-PAY-THIS-WEEK > MIN-SAL-LEVEL
           AND GROSS-PAY-THIS-WEEK < MAX-SAL-LEVEL
           PERFORM COMPUTE-NET
        ELSE
           PERFORM UNREASONABLE-SALARY
     ELSE . . .
```

Another way of creating a group label is, of course, by the paragraph-name. It should be chosen to convey the function of the paragraph.

- Whenever a series of paragraphs constitute a logical unit, then the SECTION name should be used to give a name to the function of the whole group. Remember, though, that the end of a section is signaled when another section begins. Thus if there is one section, there must be at least one additional section unless the single section is in the last physical position in the program. In general, instead of using PERFORM A THRU B it is better to give a section name to the paragraphs A through B and then say "PERFORM section-name" instead.

4 *Use similar names for similar items.*

- In most programs we encounter fields such as Name or Employee-Number that are present in an input file record, an output file record, and a report record. All three records contain similar items, and we should use a naming

convention that facilitates recognition of this similarity. One way to approach the nomenclature is to use qualification, which is discussed in Chapter 8.

```
MOVE NAME OF EMPL-SOURCE-FILE
    TO NAME OF PAY-REG-REPORT
MOVE ADDRESS OF EMPL-SOURCE REC
    TO ADDRESS OF EMPL-NEW-MAST-REC.
```

But qualification requires more extensive writing, and therefore programmers avoid qualification in their hurry to complete the program. Actually, the typing effort associated with a programming task is a very small portion of the total programming effort, but programmers tend to place undue weight on the amount of typing involved. Therefore, qualification should be considered a good approach to naming similar items and achieving good program documentation.

- Another way of naming similar items is to use a mnemonic prefix or suffix to differentiate the items. For example:

```
NAME-OMR   [OMR understood to mean Old Master Record]
NAME-RR    [RR = Report Record]
NAME-WS    [WS = Working Storage]
```

5 *Use a numerical prefix to indicate the physical order of paragraphs.*

- Structure charts allow for a two-dimensional representation of the program modules and their relationships. However, programs are written in one dimension—top to bottom, or beginning to end. In large programs, such a statement as PERFORM GROSS-CALC provides no clue as to where the paragraph GROSS-CALC is located. We may need to scan many pages of listing to find the paragraph. In order to make it easy to ''navigate'' through the program, a numerical prefix is assigned as part of a section or paragraph name and serves as the indicator of the physical location of the section or paragraph. The use of such prefixes is included in Figures 7-8 and 7-9.

Review

1 The purpose of this section has been to present the consensus of rules relating to the ______________ of structured COBOL programs.

format

2 Many of the rules are concerned with spacing and indentation, with the objective particularly being to make the programs easier to [read / write].

read

3 The use of group labels in the program and the use of similar data-names for similar items particularly [minimize writing time / enhance readability] for the program.

enhance readability

4 The physical location of paragraphs in a program can be made explicit by assigning a numerical ______________ to each paragraph name.

index

5 The rules presented in this section [should be following strictly / are general guidelines] for establishing the format for a COBOL program.

are general guidelines

PROGRAM DESIGN FOR CONTROL-BREAK LOGIC

We proceed now to apply the program design concepts discussed in this chapter to a common programming task. The task involves generating a report with *control breaks*. Figure 7-6 presents sample output from a sales report. Such a report is generated from records in an input file in which each input record contains a salesperson's name, a product number, and a sales amount.

The sample report in Figure 7-6 illustrates the concept of *control breaks* in reports. As the report is produced, we *break* the routine whenever a new product or a new salesperson is encountered. The salesperson and product fields serve as *control* fields. When we encounter a control break there are special actions taken.

When a new salesperson is encountered, the report prints a product total for the last product sold by the previous salesperson. Also, the total sales for the previous salesperson are printed before the data for the new salesperson are printed. In Figure 7-6, when the first record for saleswoman LINDA was encountered, the $170.54 total for product 345 was printed, followed by the total $395.81 for salesman JOHN. After these two totals were printed, then the first report line for LINDA was printed.

The action of printing the above totals is referred to as processing the *footing* for a control break item. Also, the action of presenting the first line for a new salesperson as differentiated from the other report lines for that salesperson is called the *heading* for that control break. Studying Figure 7-6 we can see that, similarly, we print a control heading report line for the first line of a new *product,* and the control footing for that product. When no control break occurs, then a

```
SALESMAN NAME     PRODUCT NO.     SALES AMOUNT     TOTAL SALES

ADAMSON, JOHN     123                   125.27
                                        100.00

    * TOTAL FOR PRODUCT  123                           $225.27

ADAMSON, JOHN     345                    50.00
                                        110.25
                                         10.29

    * TOTAL FOR PRODUCT  345                           $170.54

   ** TOTAL FOR SALESMAN   ADAMSON, JOHN               $395.81

ROSELLE, LINDA    123          4,000.00

    * TOTAL FOR PRODUCT  123                         $4,000.00

   ** TOTAL FOR SALESMAN   ROSELLE, LINDA            $4,000.00

WILLIAMS, MARY    123                   200.00
                                        500.00
                                        300.00
                                      1,125.00


SALESMAN NAME     PRODUCT NO.     SALES AMOUNT     TOTAL SALES
    * TOTAL FOR PRODUCT  123                         $2,125.00

   ** TOTAL FOR SALESMAN   WILLIAMS, MARY            $2,125.00

      *** TOTAL SALES FOR REPORT                     $6,520.81
```

FIGURE 7-6
SAMPLE OUTPUT FOR A PROGRAM WITH TWO CONTROL BREAKS.

detail report line is printed, which in Figure 7-6 simply consists of listing the sales amount for the (implied) salesperson and product.

There are two special cases, however. When the very first record is processed and we thus start with a new salesperson, there is no need to print control footings for the preceding product and salesperson, since there are no preceding ones. The second special case involves processing the very last data record. After printing the detail line for that record, we need to print the product and salesperson footings even though no new salesperson was encountered.

A final consideration involves the page structure of the report. At the top of each page we want to print a *page heading*. Similarly, at the bottom of a page we may want to print a *page footing*, such as a page number. In Figure 7-6 no page footing is shown, but such a footing could have been included.

Thus far in this discussion we essentially have analyzed the programming task at a general level and have identified the functions needed to implement the task. Since we have not been concerned with how to do each function, we have in fact been using the black box approach. As a result of the above analysis we can construct the structure chart in Figure 7-7, which shows the main modules and their hierarchical relationships.

Reviewing Figure 7-7, we note that "Print line" is a module subordinate to five other modules. Rather than repeating the module five times, and since it is near the bottom of the chart, we have drawn it only once for ease of reference. This "Print line" module contains the logic of where to print a line. If we are at

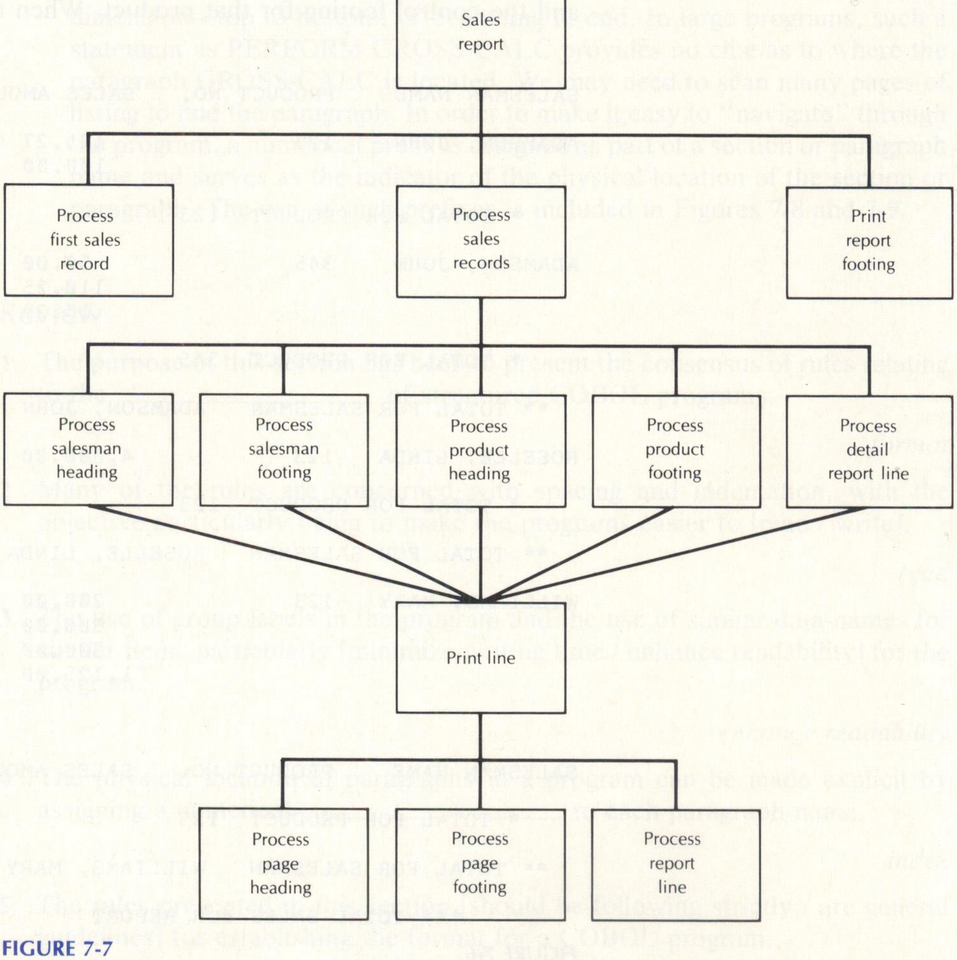


FIGURE 7-7
STRUCTURE CHART FOR CONTROL BREAK LOGIC.

the top of the page, we need to print the page heading. If we are at the bottom of the page, we need to print the page footing, if any, and then the next page heading. If we are at the middle of the page then we simply print the report line.

The structure chart in Figure 7-7 can be easily adapted to more general cases. If we had another level of control break, we could incorporate the additional break in the chart also. For example, if salespeople were grouped into departments, we would add "department heading" and "department footing" as two more modules in the chart. Designing a structure chart such as the one in Figure 7-7 provides us with a model that can be adapted and applied to specific cases.

Review

1 When an inventory report includes information and total dollar value of inventory for a given product group before the inventory for the next group is reported, the report is said to include ________________ breaks.

control

2 The printing of product-group totals in a report is referred to as being the ________________ for that control break.

footing

3 The printing of a new product-group name following the report of the total for the preceding product group is called the ________________ for that control break.

heading

4 In terms of the overall page structure of the report, the top of each page generally includes a ________________ while the bottom of each page may include a ________________.

page heading
page footing

5 An appropriate type of chart that can be used to portray the control-break logic needed to produce a report, including hierarchical relationships of the control breaks, is the ________________ chart.

structure

SAMPLE PROGRAM WITH CONTROL BREAKS

In this section we present a more detailed design for the report program whose output is illustrated in Figure 7-6. The analysis in the preceding section resulted in the rather abstract structure chart in Figure 7-7. Further analysis of the task results in the more detailed chart presented in Figure 7-8. In this chart we have numbered each module with a 3-digit prefix that corresponds to the numbering of paragraphs in the sample program in Figure 7-9. Such numbering makes it easy to cross-reference the structure chart and the program.

The process of going from the general chart in Figure 7-7 to the more specific one in Figure 7-8 involves the process of further partitioning the tasks represented by top modules. For instance, we asked the question: "What functions are needed to process the first record?" The answer to that question resulted in the three subordinate modules shown in the upper-left corner of Figure 7-8.

Reviewing Figure 7-8 notice the module labeled 030-DETERMINE-CONTROL-BREAK-LEVEL. What was the basis for including this module in the chart? We asked the question: "What is needed in order to select execution of particular

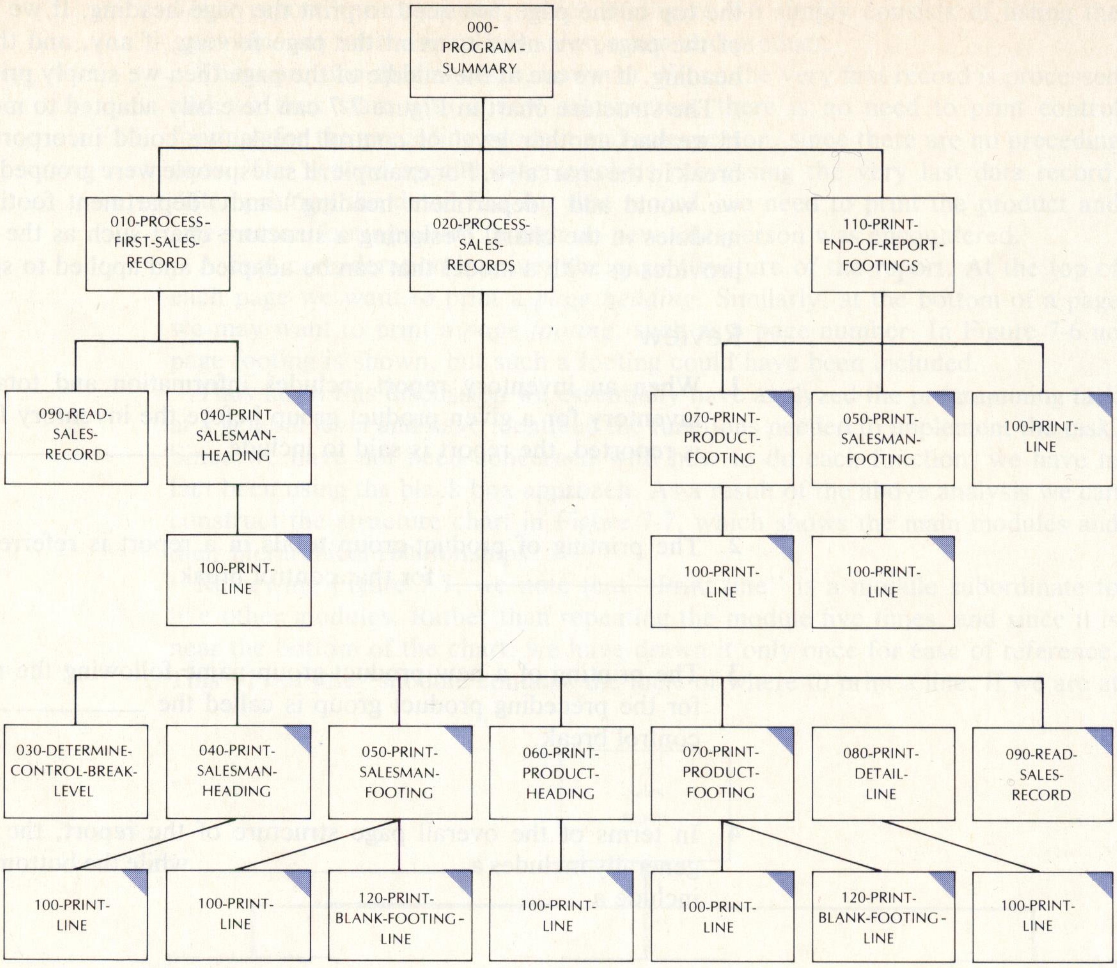


FIGURE 7-8

STRUCTURE CHART FOR SAMPLE REPORT PROGRAM.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. CONTROL-BREAKS.
*
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
*
INPUT-OUTPUT SECTION.
*
FILE-CONTROL.
    SELECT SALES-FILE  ASSIGN TO  file name/device.
    SELECT REPORT-FILE ASSIGN TO  printer name.
/
DATA DIVISION.
*
FILE SECTION.
*
FD  SALES-FILE      LABEL RECORDS ARE OMITTED
                    RECORD CONTAINS 80 CHARACTERS
                    DATA RECORD IS SALES-RECORD.

01  SALES-RECORD.
    02 SALESMAN-NAME          PIC X(15).
    02 PRODUCT-NUMBER         PIC 9(3).
    02 SALES-AMOUNT           PIC 9(4)V99.
    02 FILLER                 PIC X(56).
```

FIGURE 7-9

LISTING OF PROGRAM WITH CONTROL BREAKS.

```
*
 FD  REPORT-FILE   LABEL RECORDS ARE OMITTED
                   DATA RECORD IS REPORT-RECORD.
 Ø1  REPORT-RECORD                PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 Ø1  PREVIOUS-VALUES.
     Ø2 PREVIOUS-SALESMAN-NAME  PIC X(15).
     Ø2 PREVIOUS-PRODUCT-NUMBER PIC 9(3).
*
 Ø1  PROGRAM-FLAGS.
     Ø2 END-OF-FILE-INDICATOR    PIC XXX    VALUE 'NO'.
         88  END-OF-FILE                    VALUE IS 'YES'.
*
     Ø2 CONTROL-BREAK-LEVEL      PIC 9.
         88  NO-BREAK                       VALUE ZERO.
         88  PRODUCT-BREAK                  VALUE 1.
         88  SALESMAN-BREAK                 VALUE 2.
*
     Ø2 FOOTING-INDICATOR        PIC XXX    VALUE 'NO'.
         88  NO-FOOTING                     VALUE 'NO'.
         88  YES-FOOTING                    VALUE 'YES'.
*
 Ø1  PAGE-SIZE                   PIC 99     VALUE 25.
*
 Ø1  PAGE-LINE-COUNTER           PIC 99     VALUE 25.
 Ø1  PAGE-HEADING.
     Ø2 FILLER                 PIC X(1Ø) VALUE SPACE.
     Ø2 FILLER                 PIC X(13) VALUE 'SALESMAN NAME'.
     Ø2 FILLER                 PIC X(4)  VALUE SPACES.
     Ø2 FILLER                 PIC X(11) VALUE 'PRODUCT NO.'.
     Ø2 FILLER                 PIC X(3)  VALUE SPACES.
     Ø2 FILLER                 PIC X(12) VALUE 'SALES AMOUNT'.
     Ø2 FILLER                 PIC X(4)  VALUE SPACES.
     Ø2 FILLER                 PIC X(11) VALUE 'TOTAL SALES'.
*
 Ø1  REPORT-LINE.
     Ø2 FILLER                 PIC X(1Ø) VALUE SPACES.
     Ø2 SALESMAN-NAME-OUT      PIC X(15).
     Ø2 FILLER                 PIC X(2)  VALUE SPACES.
     Ø2 PRODUCT-NUMBER-OUT     PIC 999.
     Ø2 FILLER                 PIC X(11) VALUE SPACES.
     Ø2 SALES-AMOUNT-OUT       PIC Z,ZZ9.99.
     Ø2 FILLER                 PIC X(8)  VALUE SPACES.
     Ø2 TOTAL-SALES-OUT        PIC $$$,$$9.99.
*
 Ø1  SALESMAN-FOOTING.
     Ø2 FILLER                      PIC X(12) VALUE SPACES.
     Ø2 FILLER                      PIC X(24)
         VALUE '** TOTAL FOR SALESMAN  '.
     Ø2 SALESMAN-NAME-FOOTING       PIC X(15).
     Ø2 FILLER                      PIC X(3)  VALUE SPACES.
     Ø2 TOTAL-SALESMAN-FOOTING      PIC $$,$$$,$$9.99.
*
 Ø1  PRODUCT-FOOTING.
     Ø2 FILLER                      PIC X(14) VALUE SPACES.
     Ø2 FILLER                      PIC X(21)
         VALUE '* TOTAL FOR PRODUCT  '.
     Ø2 PRODUCT-NUMBER-FOOTING      PIC 999.
     Ø2 FILLER                      PIC X(16) VALUE SPACES.
     Ø2 TOTAL-PRODUCT-FOOTING       PIC $$,$$$,$$9.99.
*
 Ø1  REPORT-FOOTING.
     Ø2 FILLER                      PIC X(18) VALUE SPACES.
     Ø2 FILLER                      PIC X(26)
         VALUE '*** TOTAL SALES FOR REPORT'.
     Ø2 FILLER                      PIC X(1Ø) VALUE SPACES.
     Ø2 TOTAL-REPORT-FOOTING        PIC $$,$$$,$$9.99.
```

FIGURE 7-9 *(Continued)*

LISTING OF PROGRAM WITH CONTROL BREAKS.

```
*
 01  SALES-TOTALS.
     02 PRODUCT-TOTAL-SALES        PIC 9(6)V99 VALUE ZERO.
     02 SALESMAN-TOTAL-SALES       PIC 9(6)V99 VALUE ZERO.
     02 REPORT-TOTAL-SALES         PIC 9(7)V99 VALUE ZERO.
 PROCEDURE DIVISION.
*
 000-PROGRAM-SUMMARY.
     OPEN INPUT SALES-FILE.
     OPEN OUTPUT REPORT-FILE.
*
     PERFORM 010-PROCESS-FIRST-SALES-RECORD.
*
     PERFORM 020-PROCESS-SALES-RECORDS
        UNTIL END-OF-FILE.
*
     PERFORM 110-END-OF-REPORT-FOOTING.
*
     CLOSE SALES-FILE
           REPORT-FILE.
*
     STOP RUN.
/
 010-PROCESS-FIRST-SALES-RECORD.
     PERFORM 090-READ-SALES-RECORD.
     IF NOT END-OF-FILE
        MOVE SALESMAN-NAME TO PREVIOUS-SALESMAN-NAME
        MOVE PRODUCT-NUMBER TO PREVIOUS-PRODUCT-NUMBER
        PERFORM 040-PRINT-SALESMAN-HEADING
        PERFORM 090-READ-SALES-RECORD.
*
 020-PROCESS-SALES-RECORDS.
     PERFORM 030-DETERMINE-CNTRL-BRK-LEVEL.
     IF CONTROL-BREAK-LEVEL > 0
        PERFORM 070-PRINT-PRODUCT-FOOTING.
     IF CONTROL-BREAK-LEVEL > 1
        PERFORM 050-PRINT-SALESMAN-FOOTING.
     IF CONTROL-BREAK-LEVEL = 0
        PERFORM 080-PRINT-DETAIL-LINE
     ELSE
        IF CONTROL-BREAK-LEVEL = 1
           PERFORM 060-PRINT-PRODUCT-HEADING
        ELSE
           PERFORM 040-PRINT-SALESMAN-HEADING.
*
*
*  IN REVISED STANDARD WE COULD WRITE INSTEAD:
*
*  EVALUATE CONTROL-BREAK-LEVEL
*     WHEN ZERO PERFORM 080-PRINT-DETAIL-LINE
*     WHEN 1    PERFORM 060-PRINT-PRODUCT-HEADING
*     WHEN 2    PERFORM 040-PRINT-SALESMAN-HEADING.
*
*
     MOVE SALESMAN-NAME TO PREVIOUS-SALESMAN-NAME.
     MOVE PRODUCT-NUMBER TO PREVIOUS-PRODUCT-NUMBER.
     PERFORM 090-READ-SALES-RECORD.
 030-DETERMINE-CNTRL-BRK-LEVEL.
     IF SALESMAN-NAME NOT = PREVIOUS-SALESMAN-NAME
        MOVE 2 TO CONTROL-BREAK-LEVEL
     ELSE
        IF PRODUCT-NUMBER NOT = PREVIOUS-PRODUCT-NUMBER
           MOVE 1 TO CONTROL-BREAK-LEVEL
        ELSE
           MOVE 0 TO CONTROL-BREAK-LEVEL.
*
 040-PRINT-SALESMAN-HEADING.
     MOVE SPACES TO REPORT-LINE.
     MOVE SALESMAN-NAME TO SALESMAN-NAME-OUT.
     MOVE PRODUCT-NUMBER TO PRODUCT-NUMBER-OUT.
```

FIGURE 7-9 *(Continued)*

LISTING OF PROGRAM WITH CONTROL BREAKS.

```
         MOVE SALES-AMOUNT TO SALES-AMOUNT-OUT.
         PERFORM 100-PRINT-LINE.
         ADD SALES-AMOUNT TO PRODUCT-TOTAL-SALES.
*
     050-PRINT-SALESMAN-FOOTING.
         MOVE PREVIOUS-SALESMAN-NAME TO SALESMAN-NAME-FOOTING.
         MOVE SALESMAN-TOTAL-SALES TO TOTAL-SALESMAN-FOOTING.
         MOVE SALESMAN-FOOTING TO REPORT-LINE.
         MOVE 'YES' TO FOOTING-INDICATOR.
         PERFORM 100-PRINT-LINE.
         MOVE 'NO' TO FOOTING-INDICATOR.
         PERFORM 120-PRINT-BLANK-FOOTING-LINE.
         ADD SALESMAN-TOTAL-SALES TO REPORT-TOTAL-SALES.
         MOVE ZERO TO SALESMAN-TOTAL-SALES.
*
     060-PRINT-PRODUCT-HEADING.
         MOVE SPACES TO REPORT-LINE.
         MOVE SALESMAN-NAME TO SALESMAN-NAME-OUT.
         MOVE PRODUCT-NUMBER TO PRODUCT-NUMBER-OUT.
         MOVE SALES-AMOUNT TO SALES-AMOUNT-OUT.
         PERFORM 100-PRINT-LINE.
         ADD SALES-AMOUNT TO PRODUCT-TOTAL-SALES.
*
     070-PRINT-PRODUCT-FOOTING.
         PERFORM 120-PRINT-BLANK-FOOTING-LINE.
         MOVE PREVIOUS-PRODUCT-NUMBER TO PRODUCT-NUMBER-FOOTING.
         MOVE PRODUCT-TOTAL-SALES TO TOTAL-PRODUCT-FOOTING.
         MOVE PRODUCT-FOOTING TO REPORT-LINE.
         MOVE 'YES' TO FOOTING-INDICATOR.
         PERFORM 100-PRINT-LINE.
         MOVE 'NO' TO FOOTING-INDICATOR.
         PERFORM 120-PRINT-BLANK-FOOTING-LINE.
         ADD PRODUCT-TOTAL-SALES TO SALESMAN-TOTAL-SALES.
         MOVE ZERO TO PRODUCT-TOTAL-SALES.
*
     080-PRINT-DETAIL-LINE.
         MOVE SPACES TO REPORT-LINE.
         MOVE SALES-AMOUNT TO SALES-AMOUNT-OUT.
         PERFORM 100-PRINT-LINE.
         ADD SALES-AMOUNT TO PRODUCT-TOTAL-SALES.
*
     090-READ-SALES-RECORD.
         READ SALES-FILE RECORD
             AT END MOVE 'YES' TO END-OF-FILE-INDICATOR.
*
*    IN REVISED STANDARD WE COULD WRITE:
*
*    READ SALES-FILE RECORD
*        AT END SET END-OF-FILE TO TRUE.
*
*
     100-PRINT-LINE.
         IF PAGE-LINE-COUNTER > PAGE-SIZE OR
             PAGE-LINE-COUNTER = PAGE-SIZE
             WRITE REPORT-RECORD FROM PAGE-HEADING
                 AFTER ADVANCING PAGE
             MOVE SPACES TO REPORT-RECORD
             WRITE REPORT-RECORD AFTER ADVANCING 1
             MOVE 2 TO PAGE-LINE-COUNTER
*
             IF NO-FOOTING
                 MOVE SALESMAN-NAME TO SALESMAN-NAME-OUT
                 MOVE PRODUCT-NUMBER TO PRODUCT-NUMBER-OUT.
*
         WRITE REPORT-RECORD FROM REPORT-LINE
             AFTER ADVANCING 1.
         ADD 1 TO PAGE-LINE-COUNTER.
*
     110-END-OF-REPORT-FOOTING.
         PERFORM 070-PRINT-PRODUCT-FOOTING.
```

FIGURE 7-9 *(Continued)*
LISTING OF PROGRAM WITH CONTROL BREAKS.

```
           PERFORM 050-PRINT-SALESMAN-FOOTING.
           MOVE REPORT-TOTAL-SALES TO TOTAL-REPORT-FOOTING.
           MOVE REPORT-FOOTING TO REPORT-LINE.
           MOVE 'YES' TO FOOTING-INDICATOR.
           PERFORM 100-PRINT-LINE.
      *
       120-PRINT-BLANK-FOOTING-LINE.
           IF PAGE-LINE-COUNTER < PAGE-SIZE
               MOVE SPACES TO REPORT-LINE
               PERFORM 100-PRINT-LINE.
      *
```

FIGURE 7-9 *(Continued)*
LISTING OF PROGRAM WITH CONTROL BREAKS.

control headings and footings?'' We then realized that their execution is dependent on the *particular* control break. So, we concluded that a black box is needed to do the task of ''sorting out'' the control break situation.

The 100-PRINT-LINE module(s) in Figure 7-8 do not show subordinates, unlike the counterpart in Figure 7-7. Since no page footing is needed in this case, and since the page heading is a minor task, we chose to combine those functions in one module. However, should we later need to add more elaborate page heading and/or page footing functions, it would be easy to add both to the chart and to the program.

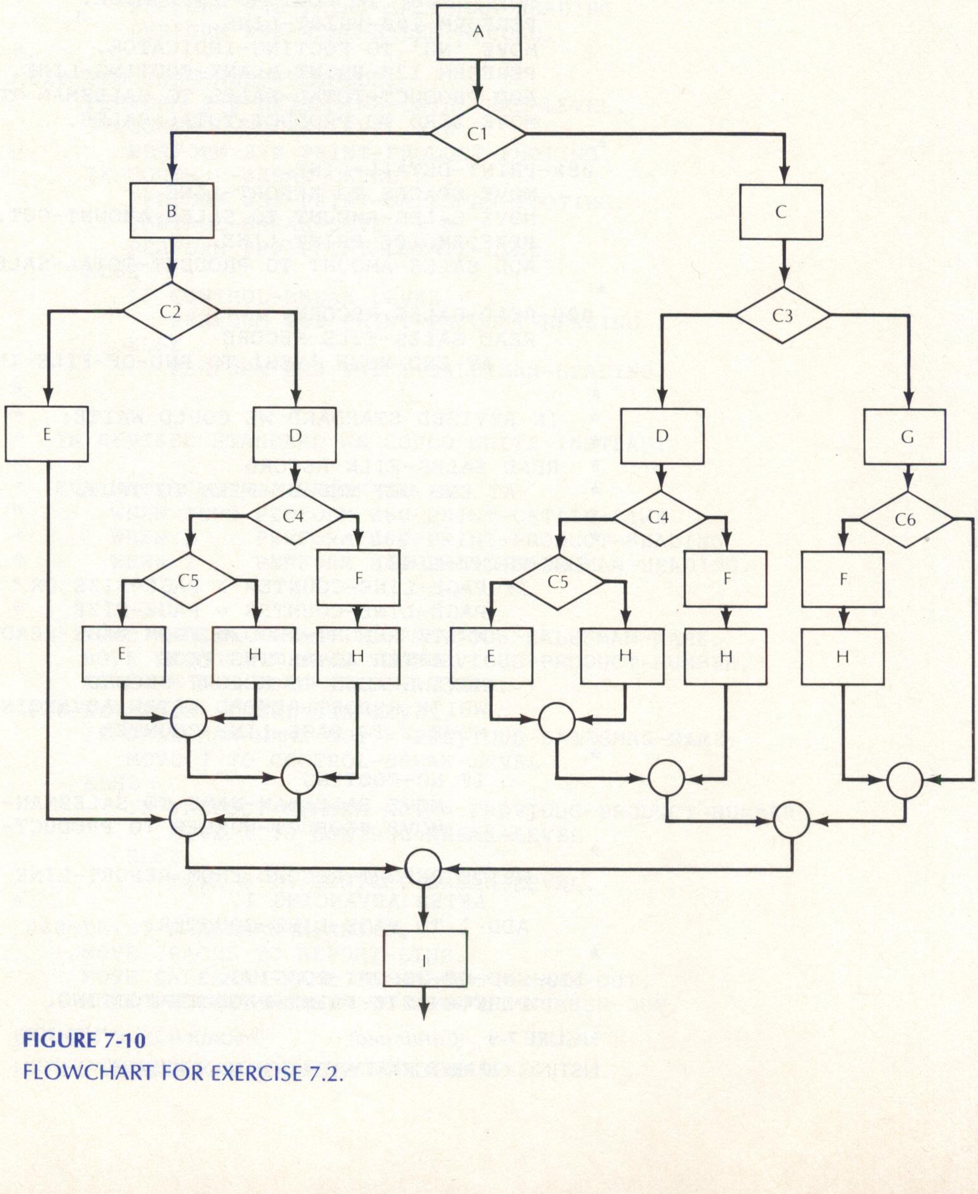


FIGURE 7-10
FLOWCHART FOR EXERCISE 7.2.

This example illustrates how we adapt a general chart such as the one in Figure 7-7 to a specific situation. This is accomplished either by adding detail, as in the case of the "Process first sales record" or abstracting from detail, as in the case of the "Print-line" module in Figure 7-7, in each case compared with Figure 7-8. A well-designed program should provide such flexibility by having two qualities:

- Each function should be cohesive and clear.
- The relationships between functions should follow the hierarchical form of one top module superior to all modules, and each subordinate module having one superior.

Programs with these qualities are relatively easy to understand and modify as needs change.

EXERCISES

7.1 Demonstrate the program design approach for a program that reads a file of student transcripts and prints a report as outlined below. There are three types of records on the file and they are so designated by a field called REC-TYPE. These records are: Name Records, Semester Records, and Course Records. The program lists each student name once and for as many semesters as are available it prints the semester identification and all the courses taken, and then computes the grade point average (GPA) for each semester. At the end, the total semester hours and the overall GPA are also given for each student. Finally, a summary report for all students is produced showing total credits and average GPA.

```
JOHN A. JOHNSON
  FALL 1988                     CREDITS       GRADE
    HISTORY                        3            B
    COBOL                          3            A
    ACCOUNTING                     3            C
    GPA   3.00
  SPRING 1984                   CREDITS       GRADE
                 .
                 .
                 .
  TOTAL CREDITS   126
    OVERALL GPA   3.25
MARY M. MURPHY
                 .
                 .
                 .
TOTAL CREDITS FOR ALL STUDENTS   XXXX
AVERAGE GPA FOR ALL STUDENTS   X.XX
```

7.2 Figure 7-10 presents a detailed flowchart segment for a program.

a Demonstrate your understanding of the concept of levels of abstraction by drawing two revised versions of the flowchart—one at a high and one at an intermediate level of abstraction.

b Draw a structure chart of the functions included in Figure 7-10. Each rectangular block in the flowchart stands for a function.

7.3 Revise the sample program in Figure 7-8 so that a third control break is included. Each input record includes a department name in columns 25–34, as well as the salesperson name in columns 1–15, the product number in columns 16–18, and the sales amount in columns 19–24.

Assume that department is a major control break. In other words, salespeople are grouped within departments and products are grouped under salespeople, as before. Include a page number at the bottom of each page.

Draw an appropriate structure chart and write a complete program.

8 ADDITIONAL LANGUAGE FEATURES

INTRODUCTION

This chapter presents a number of additional program statements. In general, these statements do certain things more easily than otherwise would be possible. As the chapter table of contents indicates, a great number of statements are presented and described. It would be difficult for a beginning COBOL programmer to incorporate all of these options in one or even several programs. Thus, a realistic objective in studying this chapter is to become aware of these additional features and to understand their functions well enough so as to refer back to them if and when a programming occasion invites their use.

IDENTIFICATION DIVISION

In the revised language the AUTHOR, INSTALLATION, DATE-WRITTEN, DATE-COMPILED, and SECURITY paragraph names in the IDENTIFICATION DIVISION are considered obsolete. These paragraphs were described briefly in Chapter 2.

Since comments can be included any place in the program, the above paragraphs are unnecessary. With the exception of the DATE-COMPILED paragraph, all others are treated as comments even in the 1974 version of the language.

ENVIRONMENT DIVISION

In the revised language the CONFIGURATION SECTION is optional. Thus we could have an abbreviated program beginning as follows:

```
IDENTIFICATION DIVISION
PROGRAM-ID. SAMPLE.
ENVIRONMENT DIVISION.
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT . . . etc.
```

Also, the INPUT-OUTPUT SECTION is optional in both the 1974 and the revised version. Omitting this section would mean that the program uses no files, a situation that may be common in subprograms, but not in main programs. Subprograms are discussed in Chapter 15.

In Figure 8-1 we present a partial format for the SPECIAL-NAMES paragraph in order to explain some additional features. Figure 8-1*a* presents the general format while Figure 8-1*b* illustrates its application by means of some examples.

Implementor-name in the figure is a name which has meaning for a particular compiler. In the example, CONSOLE, SYSIN, and C12 are assumed to be implementor-names for a particular compiler. The *mnemonic-names* provide for the opportunity to create names that are meaningful, such as CRT. Such mnemonic names may then be used with the ACCEPT, DISPLAY, and WRITE verbs as illustrated in the following examples.

```
DISPLAY 'HELLO, WHAT IS YOUR NAME' UPON CRT
ACCEPT STARTING-CHECK-NO FROM READER
WRITE REPORT-LINE AFTER ADVANCING BOTTOM-OF-PAGE.
```

Note that the implementor-names are specific to each compiler, so you will have to consult the appropriate reference manual before using such features.

The ALPHABET clause allows the user to name one of the three choices

```
ENVIRONMENT DIVISION.
[CONFIGURATION SECTION.]
[SOURCE-COMPUTER.]
[OBJECT-COMPUTER.]
[SPECIAL-NAMES.]
  [implementor-name-1  IS mnemonic-name]

                                     { STANDARD-1 }
  [ALPHABET  alphabet-name-1  IS     { STANDARD-2 }
                                     { NATIVE     }
  [SYMBOLIC CHARACTER  {symbolic-character-1  IS  integer-1
    IN  alphabet-name-2} . . . ] ]
  [CURRENCY SIGN IS literal]
  [DECIMAL-POINT IS COMMA]
```

(*a*) Format

```
SPECIAL-NAMES.
     CONSOLE IS CRT
     SYSIN IS READER
     CI2 IS BOTTOM-OF-PAGE
     ALPHABET ASCII IS STANDARD-1
     SYMBOLIC CHARACTER BEEP IS 7 IN ASCII
     CURRENCY SIGN IS 'F'
     DECIMAL-POINT IS COMMA
```

(*b*) Sample Uses

FIGURE 8-1
SOME SPECIAL-NAMES OPTIONS.

shown in Figure 8-1. STANDARD-1 refers to the American National Standard Code X3.4-1977 for Information Interchange. STANDARD-2 refers to the International Standard 646 Code for Information Processing Interchange. Finally, NATIVE refers to the code that is native to that computer, such as the EBCDIC code used in most IBM systems. Such codes specify the collating sequence of characters as briefly explained in Chapter 6 in the section on relation conditions.

Refer to the following program segment included in Figure 8-1*b*:

```
ALPHABET ASCII IS STANDARD-1
SYMBOLIC CHARACTER BEEP IS 7 IN ASCII
```

The SYMBOLIC CHARACTER specification uses the alphabet-name ASCII, as defined by ALPHABET ASCII IS STANDARD-1. If the character-value 7 in the collating sequence of STANDARD-1 causes the terminal bell to ring, the following program statements would cause the bell to ring:

```
01  DISPLAY-FIELD  PIC X.
MOVE BEEP TO DISPLAY-FIELD.
DISPLAY DISPLAY-FIELD . . .
```

The last two options in the format in Figure 8-1*a* refer to COBOL provisions for international usage. Changing the dollar sign and the convention of using a comma in lieu of a decimal point can be accommodated by means of two special clauses. These clauses are written in the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION. It might seem inappropriate to introduce them at this point, but actually they are intrinsically related to the DATA DIVISION. The dollar sign of course is not the currency symbol for other nations. The programmer may specify the currency sign using the CURRENCY clause.

For example, suppose that F is the currency sign. Then we would write CURRENCY SIGN IS 'F', and in PICTURE clauses we would use 'F' in place of $. The currency sign cannot be chosen from the following: 0 through 9, A, B, C, D, L, P, R, S, V, X, Z, space, *, +, −, comma , . , (,), ", / , or = .

In many countries outside the United States, the function of the decimal point and the comma are reversed. Thus, in Europe the numbers 1,35 and 2.534,99 are the equivalent of the American 1.35 and 2,534.99, respectively. To accommodate these different conventions we use the DECIMAL-POINT IS COMMA clause. Once this clause has been used, the function of comma and period are exchanged in the character string of the PICTURE clause and in numeric literals. For example, it is correct to write:

```
02  AMOUNT        PIC Z.ZZZ.ZZZ,ZZ
```

since the two decimal points have the same function as commas.

Review

1 In the SPECIAL-NAMES paragraph of the ENVIRONMENT DIVISION an implementor-name, such as for a CRT, has meaning only with respect to the particular ____________ being used.

compiler

2 In the SPECIAL-NAMES paragraph, the type of name that is created by the user to be a meaningful name for an implementor, such as a CRT, is called a ____________-name.

mnemonic

3 The clause in the SPECIAL-NAMES paragraph that allows the user to specify the collating sequence of characters is the ____________ clause.

ALPHABET

4 The dollar sign can be changed to the appropriate symbol for another monetary system by use of the ________________ clause, in the SPECIAL-NAMES paragraph.

CURRENCY

5 In monetary systems in which the decimal point and comma have functions opposite to those recognized in the United States, the interchange of these symbols can be achieved by use of the ________________ clause.

DECIMAL-POINT

THE USAGE CLAUSE

Numeric data in a computer may be represented in one of two basic modes. They may be represented as character data or as numeric data. Both modes utilize binary characters (bits) of zero and one, but the meaning of a given bit depends on the coding method used.

In character mode, each decimal digit in a number is represented by a group of binary bits. Commonly, either six or eight bits comprise each digit; thus, in the six-bit Binary Coded Decimal (BCD) form, the decimal number 19 is represented as 000001001010. The first six bits represent the decimal digit 1, and the last six bits the digit 9. We also should mention two other commonly used coding schemes: the American Standard Code for Information Interchange (ASCII) and the Extended Binary Coded Decimal Interchange Code (EBCDIC), the latter used primarily by IBM.

Numeric data in numeric mode consists of binary bits that have positional values analogous to the decimal system. In the decimal system, the first digit in the number 111 has meaning (100), entirely different from that of the third digit (1), even though they look alike. The string of binary bits 000001001010 viewed as a binary number signifies a quantity of 74, not 19, as it would in the BCD code described above. Actually, a variety of numeric coding schemes are in use. We mention their names for reference and direct the interested reader to other sources for explanation (most introductory EDP texts contain descriptions of such codes). The common coding forms are: fixed-point binary, floating-point of single or double precision, packed-decimal, and zoned-decimal.

The arithmetic registers of computers perform arithmetic with numeric data that is in numeric, not character, mode. If numeric data is represented in character mode, it must first be converted to numeric mode before arithmetic computations can be performed.

In COBOL, data in character mode is described as being in DISPLAY mode, while data in numeric mode is described as being in COMPUTATIONAL mode. DISPLAY is the default condition: all data items are assumed to be in DISPLAY mode unless they are declared to be COMPUTATIONAL. The declaration is done in the DATA DIVISION with the USAGE clause. The general format is:

```
            ⎧ BINARY          ⎫
            ⎪ COMPUTATIONAL   ⎪
[USAGE is   ⎨ COMP            ⎬ ]
            ⎪ DISPLAY         ⎪
            ⎪ INDEX           ⎪
            ⎩ PACKED-DECIMAL  ⎭
```

Consider the following examples:

```
02  A   PIC 99.
02  B   PIC 99 USAGE DISPLAY.
02  C   PIC S9(4)V99 USAGE COMPUTATIONAL.
```

```
02 D PIC 99 USAGE COMP.
02 E PIC S9(8) USAGE BINARY.
02 F PIC S9(4)V99 USAGE PACKED-DECIMAL.
```

The first example omits the USAGE clause, and therefore the item will be in DISPLAY mode by default. The second example makes the declaration of USAGE DISPLAY explicit. The third and fourth examples show use of COMPUTATIONAL in full and abbreviated form. The last two examples illustrate options that are available in the revised language for binary and packed decimal representations (No example of INDEX usage is shown because this subject is covered in Chapter 11, ''Additional Table-Handling Features''.)

As a general comment, use of the COMPUTATIONAL, BINARY, and PACKED-DECIMAL options increases the execution speed of arithmetic operations. When numeric data is in DISPLAY form it needs to be converted to some other form before arithmetic is performed. Then the arithmetic result is converted back to DISPLAY form before storage in the appropriate field. The reason for such conversions is that the hardware does not do arithmetic with data in DISPLAY form. Each machine uses one or more ''computational'' representations for its arithmetic operations. The idea of the USAGE option, then, is to allow the programmer to specify a more efficient representation, depending on the machine and the intended use. In general, the COMPUTATIONAL option is the one expected to fit the computational mode of a given machine. Still, the BINARY and PACKED-DECIMAL allow more specific choices with respect to the computational code used. For example, when doing table-handling operations subscript values may be more efficiently handled in binary form. Also, IBM systems tend to use packed-decimal form; therefore a data file from an IBM system may have data stored in such form and we could use the data appropriately only by specifying PACKED-DECIMAL

While execution efficiency is always important, in terms of today's costs and speeds it hardly seems worth the effort worrying about the inefficiency of converting data from DISPLAY to some computational form and back. Typical COBOL programs generally execute only a few hundred thousand arithmetic operations. Therefore, the increase in speed of execution may be so small that it is not worth the trouble of worrying about USAGE. Nevertheless, use of the appropriate USAGE will improve efficiency, and in programs that do a lot of arithmetic (many millions of operations), the choice may be worthwhile.

Data entered via an online keyboard terminal are in DISPLAY mode, and thus no COMPUTATIONAL fields should be used in such record descriptions. Similarly, printing COMPUTATIONAL data makes no sense. However, COMPUTATIONAL data may be read from or written onto magnetic media, such as tapes and disks.

Most compilers specify different forms of COMPUTATIONAL. For instance, many systems use COMPUTATIONAL-1 (or COMP-1) to define a single precision floating-decimal field. ANSI COBOL recognizes only one form, COMPUTATIONAL (or COMP), and it is up to the implementor of the language to define the specific meaning of this standard term. Unless compelling reasons lead to the choice of nonstandard options, such as COMP-1 or the like, the programmer will find it advantageous in the long run to stick with the standard.

It should be noted that moving zeros to a group item whose elementary items contain USAGE COMPUTATIONAL clauses may result in nonzero data. Consider this group item:

```
01 GROUP-ITEM.
   02 AMOUNT-1    PIC 99V99 USAGE COMPUTATIONAL.
   02 AMOUNT-2    PIC 999V99 USAGE COMPUTATIONAL.
```

A statement such as MOVE ZEROS to GROUP-ITEM will move ''character'' zeros into GROUP-ITEM. A zero in character mode is different from a zero in

numeric mode. Therefore, if we subsequently write ADD TOTAL TO AMOUNT-1, erroneous results will be obtained from the arithmetic operation, since AMOUNT-1 does not contain a zero value. To avoid such problems, we should move zeros to each individual numeric field that has been defined as USAGE COMPUTATIONAL. Thus, MOVE ZERO TO AMOUNT-1, AMOUNT-2 would be the appropriate instruction for setting these two fields equal to zero. Of course, in the absence of the USAGE COMPUTATIONAL clauses, MOVE ZERO TO GROUP-ITEM would have been no problem in this case.

Review

1 Numeric data stored in a computer may be represented in one of two basic modes: as ________ data or as ________ data.

character; numeric

2 In order to perform arithmetic computations, numeric data must be in ________ mode.

numeric

3 In COBOL, data in the character mode is described as being in ________ mode and data in the numeric mode is described as being ________.

DISPLAY; COMPUTATIONAL

4 In COBOL, the mode of a data item is stipulated in the ________ clause in the DATA DIVISION.

USAGE

5 When the USAGE clause is not used, the field automatically is defined as being [DISPLAY / COMPUTATIONAL] in form.

DISPLAY

6 The USAGE BINARY and USAGE PACKED-DECIMAL available with the revised standard make it possible to achieve the execution of arithmetic operations [more accurately / more efficiently].

more efficiently

7 If zeros are moved into a group item, such zeros will always be in the [character / binary numeric] mode.

character

THE SIGN CLAUSE

Use of the S PICTURE character specifies a signed field. The operational sign is stored as part of the field. Its specific form depends on the USAGE of the field. If USAGE IS DISPLAY, then the sign is typically stored in the rightmost byte of the field and it ''co-exists'' with the last digit. For example, assuming 8-bit bytes, the first four bits of the last byte may be used to store the arithmetic sign while the last four bits are used to store the last digit of the field. In COMPUTATIONAL, BINARY, and PACKED-DECIMAL representations the specific computer and option used determine the method by which the sign associated with the value is stored.

When the usage of an item is DISPLAY, then we can use the SIGN clause to specify either the location of the algebraic sign or whether it is to be stored as a separate character, or both.

The general format for the SIGN clause is:

[SIGN IS] {LEADING | TRAILING} [SEPARATE CHARACTER]

If we write:

```
02 AMOUNT-A PICTURE S999 SIGN IS LEADING.
02 AMOUNT-B PICTURE S999 SIGN IS TRAILING.
```

we are specifying that these two signed fields contain the operational sign in the first and in the last digit position, respectively. Thus, if they both contained the numeric value −243 and the character K happened to represent −2 while L happened to represent −3, the contents of these fields would be shown as follows:

AMOUNT-A		
K	4	3

AMOUNT-B		
2	4	L

As we stated above, the convention is to store the sign in the rightmost digit position; therefore, the absence of the SIGN clause defaults to that case. The choice of LEADING rather than TRAILING is difficult to justify, but the option is available.

When the SEPARATE CHARACTER option is used, then the operational sign *is* actually represented as a separate leading or trailing character and requires a separate storage position. Consider these examples:

```
02 AMOUNT-A    PIC  S999  SIGN IS LEADING SEPARATE CHARACTER.
02 AMOUNT-B    PIC  S999  SIGN IS TRAILING SEPARATE.

MOVE   15 TO AMOUNT-A
MOVE −156 TO AMOUNT-B
```

After these MOVE instructions are executed the contents of these two fields will be:

AMOUNT-A			
+	0	1	5

AMOUNT-B			
1	5	6	−

Notice that each field consists of four positions. In AMOUNT-A the + sign is inserted as the first character. A field containing S always contains a sign, whether positive or negative. The leading zero, of course, was inserted by the MOVE so that all the characters in the field are numeric digits. In the case of AMOUNT-B the sign is negative and it is trailing.

Fields containing the SIGN clause must include the S character and are considered numeric. The system treats the sign as part of the field in MOVE operations. A MOVE from an S99 field to S99 SIGN IS . . . SEPARATE will be suitably converted by the system to change the sign representation. Similarly, reversing the sending and receiving fields in the above example will also produce conversion to the appropriate sign representation.

When the SIGN . . . SEPARATE clause is used, then source data may be entered with a sign. However, care must be taken to put the sign in the correct place and to use numeric values only, as always. For the field

```
AMOUNT-A  PIC  S9999  SIGN  LEADING  SEPARATE
```

the following data would be correct: +0010, +1234, −1000, −0001, +0000. Notice that a sign must always be present, even to represent a (positive) zero.

Review

1 If the stored values in a field could be negative, typically because subtraction is performed with respect to the numeric field, then the PICTURE clause for the field should include the ________________ PICTURE character as the [leftmost / rightmost] character.

S; leftmost

2 When describing the field size, the S character is not counted in the size of the field because typically the [leftmost / rightmost] digit position is used for a combination of digit and sign representation.

rightmost

3 If negative values are input by means of a keyboard terminal, both a digit and a negative sign must be included in the [first / last] position in the field.

last

4 The input of negative values in conjunction with COBOL programs is relatively [frequent / rare].

rare

5 When it is necessary to describe the position of the operational sign for a field explicitly, the ________________ clause is included with the data description.

SIGN

6 Fields containing the SIGN clause [must / need not] include the S character and [must / need not] be described by 9 PICTURE characters.

must; must

THE SYNCHRONIZED CLAUSE

The USAGE COMPUTATIONAL option increases program running efficiency, but it does not do the whole job. The SYNCHRONIZED clause is needed to achieve additional execution efficiency. The need for the SYNCHRONIZED clause derives from the fact that COBOL is a general language, but computers of different manufacturers differ in structure. Consider this example:

```
01  FIELD-A.
    02  FIELD-B    PIC 9 USAGE COMP.
    02  FIELD-C    PIC 99 USAGE COMP.
```

In a straightforward fashion, we defined two numeric computational fields of one- and two-decimal digits, respectively. In word-oriented computers, such as many UNIVAC, Honeywell, and CDC machines, both fields are stored in the same word, and arithmetic computations are not as efficient as when each field is stored in a separate word. In character-oriented computers, such as many IBM machines, arithmetic execution may not be as efficient as it could be because the fields are not in proper storage boundaries. The SYNCHRONIZED clause can be used to improve efficiency in this simple example.

Notice the SYNC abbreviation, which is an available alternative:

```
01  FIELD-A.
    02  FIELD-B    PIC 9 USAGE COMP SYNCHRONIZED.
    02  FIELD-B    PIC 99 USAGE COMP SYNC.
```

The use of this option is not always this simple. For other variations, one should consult the manufacturer's manual for the computer being used.

We hasten to add that the SYNCHRONIZED option should be used sparingly. In order to achieve efficiency, one must understand clearly the storage structure of a given computer and the compiler's implementation of the SYNCHRONIZED option. Because this storage structure differs among computers, program compatibility across different computers may be lost by use of the SYNCHRONIZED clause. The interested reader should study the manufacturer's manual for her or his own computer before using synchronization, and should consider use of the option only for programs that perform extensive numerical computation.

We digress for a moment to draw a brief comparison with the popular FORTRAN and BASIC languages. A user of either of these languages may wonder why COBOL needs complications such as the SYNCHRONIZED clause, while FORTRAN and BASIC do beautifully without any such considerations. These languages achieve simplicity by limiting options and forcing uniformity. For instance, all numeric integer fields in BASIC occupy a full word (in a word-oriented machine), regardless of desired field size. In contrast, COBOL allows the programmer to define integer fields from 1 to 18 digits in size. In this case, the price of flexibility is complexity.

Review

1 When the USAGE clause is not used, the field automatically is defined as being [DISPLAY / COMPUTATIONAL] in form.

DISPLAY

2 The option that can be used with the USAGE COMPUTATIONAL specification that improves the efficiency of arithmetic execution by assignment of appropriate storage boundaries is the ________________ clause.

SYNCHRONIZED

3 Use of the SYNCHRONIZED clause [could / would never] result in loss of program compatibility across different computers.

could

THE JUSTIFIED RIGHT CLAUSE

This option is used with elementary alphabetic or alphanumeric items only, and its effect is to override the convention of left-justifying nonnumeric data. Suppose we have the record description 02 TITLE PICX(10). If we write MOVE "JONES" to TITLE, the effect in TITLE will be |J|O|N|E|S| | | | | |, with the name left-justified. However, if in the DATA DIVISION we had written 02 TITLE PICTURE X(10) JUSTIFIED RIGHT, execution of the above MOVE instruction would result in | | | | | |J|O|N|E|S| in TITLE.

As indicated by this example, the JUSTIFIED RIGHT clause is always used in conjunction with the PICTURE clause for elementary items; however, it cannot be used with level 66 or level 88 items. Level 66 is a special-purpose level explained later in this chapter. In addition to arranging right-justification, the JUSTIFIED RIGHT clause also affects truncation. Without the JUSTIFIED RIGHT clause, truncation takes place from the right for alphabetic and alphanumeric data. When the JUSTIFIED RIGHT clause is used, truncation takes place from the left, as for numeric data.

Review

1 Elementary alphabetic or alphanumeric items can be positioned in the rightmost portion of the field by use of the ______ clause in the DATA DIVISION.

JUSTIFIED RIGHT

2 When the JUSTIFIED RIGHT clause is used, truncation of the content of a data item that is moved to a field that is too small takes place from the ______.

left

QUALIFICATION

Up to this point in the book, we always have indicated that every data-name must be unique in a given program. This requirement will be modified now by introducing the use of qualifiers, which retain the concept of unique data-names but expand the form to provide greater flexibility. A qualifier is a data-name of higher hierarchical level than the name it qualifies. The use of qualifiers results in having unique data-names for names that otherwise would not be unique, thus providing more flexibility in the assignment of data-names in the program. The following DATA DIVISION segment is an example of a case in which qualification would be necessary in the PROCEDURE DIVISION:

```
02  WEEKLY-TOTALS.
    03  HOURS            PICTURE 99V9.
    03  DEPARTMENT-NO    PICTURE 9(5).
02  MONTHLY-TOTALS.
    03  HOURS            PICTURE 999V9.
    03  (etc.)
```

Notice that HOURS seems to be used twice, but with respect to two different items, namely, the total hours for the week and the total hours for the month. If reference were made simply to HOURS, it would not be clear which storage field should be used; however, the use of qualifiers results in unique data-names and could be accomplished as follows in the PROCEDURE DIVISION instructions:

```
MOVE HOURS OF WEEKLY-TOTALS TO . . .
MOVE HOURS IN MONTHLY-TOTALS TO . . .
```

OF and IN in these program statements are equivalent, and use of either word after a data-name serves to signal the use of a qualifier. Since a data-name that is not unique in the program must always be qualified, the use of nonunique names always results in longer statements in the PROCEDURE DIVISION. Despite this disadvantage, qualifiers are often used because they improve documentation. For instance, the statements in the above example make it quite clear that we are working with weekly and monthly hours, respectively.

A common use of qualifiers is with records that have some fields in common, such as master and transaction records. If an employee has an assigned identification number that is included in a master record as well as in a transaction record, the following type of instruction can be included in the PROCEDURE DIVISION:

```
IF EMPLOY-NUMBER IN MASTER-RECORD EQUAL EMPLOY-NUMBER IN TRANSACTION-
    RECORD . . .
```

Again, the documentation aspect of the program is enhanced in this example in that it is quite clear what is being compared. A further use of the concept of

nonunique data-names relates to the CORRESPONDING option, which is discussed in a subsequent section of this chapter.

At times, qualification requires several qualifiers. Consider the following example:

```
01  OLD-RECORD.                          01  NEW-RECORD.
    02  TODAYS-DATE . . .                    02  TODAYS-DATE . . .
        03  MONTH . . .                          03  MONTH . . .
        03  YEAR . . .                           03  YEAR . . .
    02  LAST-PERIODS-DATE . . .              02  LAST-PERIODS-DATE . . .
        03  MONTH . . .                          03  MONTH . . .
        03  YEAR . . .                           03  YEAR . . .
        03  TOTAL . . .                          03  TOTAL . . .
```

In this example, the OLD-RECORD and NEW-RECORD are assumed to be in the same program. Notice that there are four fields named MONTH. Thus, a qualifier such as MONTH OF LAST-PERIODS-DATE does not provide a unique reference because there are two such fields, one in the OLD-RECORD and one in the NEW-RECORD. Therefore, two qualifiers are needed in order to reference a unique field such as:

$$\text{MONTH} \begin{Bmatrix} \text{OF} \\ \text{IN} \end{Bmatrix} \text{LAST-PERIODS-DATE} \begin{Bmatrix} \text{OF} \\ \text{IN} \end{Bmatrix} \text{OLD-RECORD}$$

Since the TOTAL field in the preceding program example occurs only once in each record, only a single qualifier is required; therefore, TOTAL IN OLD-RECORD is an adequate reference in this case. The use of two qualifiers, such as TOTAL OF LAST-PERIODS-DATE IN OLD-RECORD is acceptable but unnecessary for the purpose of unique identification.

Review

1 The use of qualifiers makes it possible to use the same data-name for variables that otherwise would have different data-names assigned to them. The main advantage of using qualifiers is that ______________ thereby is improved.

documentation (or interpretation of the program, etc.)

2 When a qualifier is used, it always [precedes / follows] the referenced data-name, and its use is signaled by one of two words: ______________ or ______________.

follows; OF; IN

3 Qualifiers frequently are used when a master file record is updated, using data of individual transactions. Assume that the record-name for the master file is MASTER-RECORD and the record-name of the transactions file is TRANSACTION-RECORD. Both records contain a field called CUSTOMER-NUMBER. To determine that we are dealing with two records of the same customer, we say: IF CUSTOMER-NUMBER OF ______________ EQUAL CUSTOMER-NUMBER OF ______________.

MASTER-RECORD;
TRANSACTION-RECORD
(either order)

4 A sufficient number of qualifiers must be used to differentiate a particular data field from all other data fields in the program that are identified by the same data-name. Suppose that a qualifier is used with a data-name that is unique and thus requires no qualifier. From the standpoint of programming

requirements, the qualifier is unnecessary [and the program will terminate / but will not affect program execution].

but will not affect program execution (again, such a qualifier might be used to improve documentation)

MULTIPLE DATA RECORDS

The following option is available in the file description to indicate the existence of more than one type of data record in the file:

```
FD file-name LABEL clause . . .

     { RECORD IS   }
DATA {             } record-name-1 [record-name-2] . . .
     { RECORDS ARE }
```

Consider an example. Suppose that a bank's customers may make three kinds of transactions: deposits, withdrawals, and change of address. A record containing information about a deposit or withdrawal has a different format than one pertaining to a change of address. We assume that all records identify the account by a five-digit number in the first five columns and the type of transaction by a transaction code in column 6. Then, if it is a deposit or withdrawal record, the amount is recorded in columns 7–12, while if it is a change of address the new address is recorded in columns 7–50. We can write the following data entries:

```
FD TRANSACTION-FILE
   LABEL RECORDS OMITTED
   DATA RECORDS ARE FINANCIAL-REC
                    ADDRESS-REC.
01 FINANCIAL-REC.
   02 ACCOUNT         PIC 9(5).
   02 TRANS-CODE      PIC 9.
   02 AMOUNT          PIC 9(4)V99.
   02 FILLER          PIC X(38).
01 ADDRESS-REC.
   02 ACCOUNT         PIC 9(5).
   02 TRANS-CODE      PIC 9.
   02 NEW-ADDRESS     PIC X(44).
```

In this example, the FD entry has specified two types of data records, named FINANCIAL-REC and ADDRESS-REC. It is important to emphasize that, physically, a record in this file will consist of 50 characters of data (the sum of all PIC clauses in either record description). Use of two record descriptions simply allows us to reference that data by different names and in different ways.

Both record names reference the entire 50 columns of data. Thus, MOVE FINANCIAL-REC or MOVE ADDRESS-REC do exactly the same thing: they MOVE these 50 columns of data. Similarly, ACCOUNT of FINANCIAL-REC and ACCOUNT of ADDRESS-REC refer to the same first five columns of data, as do the TRANS-CODE names. However, AMOUNT refers to the data in columns 7–12 and, according to the PIC clause, it is assumed that the data are numeric. It should be recalled that a PICTURE clause simply specifies the storage allocation to data, and not the actual contents. The actual contents come about through input or MOVE type operations. Thus, if we were dealing with a change-

of-address transaction and columns 7–12 contained the first six characters of the new address, a statement such as ADD 1 TO AMOUNT would produce unpredictable results, since we are performing arithmetic with nonnumeric data. More appropriately, we first would test to see what type of record we actually had before referencing the data in question. For this purpose, assume that we use a code of 1 for a deposit, 2 for withdrawal, and 3 for change of address. We could write:

```
IF TRANS-CODE OF FINANCIAL-REC = 1
    PERFORM PROCESS-DEPOSIT.
```

This statement checks to see if column 6 contains the value 1. Notice that, if we had used IF TRANS-CODE OF ADDRESS-REC = 1, it would be exactly the same thing, since both refer to column 6. To further clarify this point, we could have used this data description for ADDRESS-REC:

```
01  ADDRESS-REC.
    02  FILLER          PIC X(6).
    02  NEW-ADDRESS     PIC X(44).
```

ACCOUNT and TRANS-CODE need no qualification now, since they are unique names in FINANCIAL-REC. Then in the PROCEDURE DIVISION we could write:

```
IF TRANS-CODE = 1
    PERFORM PROCESS-DEPOSIT.
```

Use of TRANS-CODE without qualification refers to column 6 of the data. Beginning students often equate the generic data-name FILLER with blank spaces. This should be a good point to discard any such misconception. In this example, the FILLER in ADDRESS-REC refers to the first six columns, which we know will contain data—the account number and the transaction code.

In general, a file may consist of more than one type of data record. As a rule, there should be a field that designates the type of record involved. In our example we used TRANS-CODE as a field that was in a fixed location no matter what the record type. Then we tested the value of that field to ascertain the type of record. In general, this identifying field should be common to all record types so that, no matter what the data are in other fields, this field can be tested.

Review

1 When there is more than one type of data record in a file, the file description in the DATA DIVISION should identify [only one / more than one] file-name.

only one

2 When there is more than one type of data record in a file, the file description in the DATA DIVISION should identify [only one / more than one] record-name.

more than one

3 A coded entry in a specified field serves to differentiate the different input records when there is more than one type of data record. Particularly when the records are of variable length, the differentiating field should be located in the [left / right] part of the record field.

left

4 In the case of multiple-type record files, at any given time the internal storage can contain [only one / more than one] type of record.

only one

THE REDEFINES CLAUSE

The REDEFINES clause can be used to allow the same storage location to be referenced by different data-names or to allow a regrouping or different description of the data in a particular storage location. The general format associated with the use of this option is:

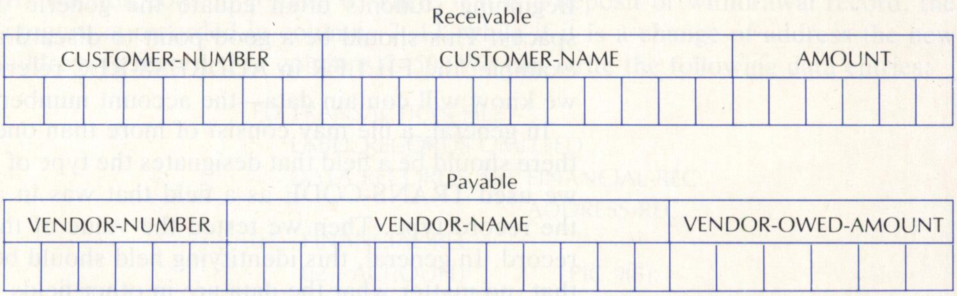
```
level-number  data-name-1 REDEFINES data-name-2
```

The following example illustrates use of this option:

```
01  SAMPLE.
    02  RECEIVABLE.
        03  CUSTOMER-NUMBER          PICTURE 9(8).
        03  CUSTOMER-NAME            PICTURE X(11).
        03  AMOUNT                   PICTURE 9(4)V99.
    02  PAYABLE REDEFINES RECEIVABLE.
        03  VENDOR-NUMBER            PICTURE 9(6).
        03  VENDOR-NAME              PICTURE X(12).
        03  VENDOR-OWED-AMOUNT       PICTURE 9(5)V99.
```

In this example, use of the REDEFINES option allows the data-names RECEIVABLE and PAYABLE to refer to the same 25 positions in internal storage.

The format of these two data items in internal storage can be portrayed as follows:



In this example, notice that the format of the data items was also changed by the use of the REDEFINES option, but that the overall size of the item was not changed.

It should be made clear that the redefinition applies to the storage area involved and not to the data that may be stored in that area at any point in time. The programmer is responsible for providing the necessary program logic so that correct reference is made to the actual data stored. In the previous example program, if we write ADD VENDOR-OWED-AMOUNT TO . . . , the result will be to add the contents of the last seven storage positions, whatever these contents might be.

There are certain conditions under which the REDEFINES clause cannot be used. Two such conditions are:

1. The REDEFINES clause cannot be used at the 01 level in the FILE SECTION. Recall that the use of multiple data records in the FD entry has the same effect as use of the REDEFINES option, in that it permits use of the same storage location for different records.
2. The REDEFINES clause cannot be used when the levels of data-name-1 and data-name-2 are different. Further, the level number must not be at the 66 or 88 level. The special-purpose 66 level is explained in the following section.

Review

1 The same storage location can be used in conjunction with two different data-names by use of the ______________ clause.

REDEFINES

2 When the REDEFINES option is used, the format of the data item [can / cannot] be changed as well.

can

3 Generally, the REDEFINES clause can be used when the two data items have the same level number. The exceptions are when the special-purpose 66 or 88 level numbers are used and when the level number is at 01 in the ______________SECTION, in which case the REDEFINES clause cannot be used.

FILE

THE RENAMES CLAUSE

The RENAMES clause provides the programmer with the capability of regrouping elementary data items. In a sense, it resembles the REDEFINES clause, except that it can form a new grouping of data items that combines several items. Use of the RENAMES clause is always signaled by the special 66 level number. The general format is:

```
66 data-name-1 RENAMES data-name-2 [THRU data-name-3]
```

Consider the following example, which includes use of the RENAMES clause.

```
01  TAX-RECORD.
    02  SOC-SEC-NUMBER          PICTURE X(9).
    02  NAME.
        03  FIRST-NAME          PICTURE X(10).
        03  INITIAL             PICTURE XX.
        03  LAST-NAME           PICTURE X(15).
    02  TOTALS-YEAR-TO-DATE.
        03  GROSS-PAY           PICTURE 9(8)V99.
        03  NET-PAY             PICTURE 9(8)V99.
        03  FED-TAX             PICTURE 9(6)V99.
        03  STATE-TAX           PICTURE 9(4)V99.
66  LAST-GROSS RENAMES LAST-NAME THRU NET-PAY.
```

Schematically, the regrouping of data fields by use of the RENAMES clause in the last statement can be portrayed as follows:

	NAME			TOTALS-YEAR-TO-DATE			
SOC-SEC-NUMBER	FIRST-NAME	INITIAL	LAST-NAME	GROSS-PAY	NET-PAY	FED-TAX	STATE-TAX
			LAST-GROSS				

In the example, LAST-GROSS is a storage field that consists of the LAST-NAME, GROSS-PAY, and NET-PAY fields. In this way, we can make reference to those three fields as one group, which would not be possible without use of the RENAMES clause or a suitable use of one or more REDEFINES clauses.

There are other rules governing use of the RENAMES option, but they are outside of the scope of this text. The user should be aware that knowledge of additional rules may be necessary for the successful use of the RENAMES option in more complex situations.

Review

1 Elementary data items that are part of different storage fields can be regrouped and formed into a new field by use of the ________________ clause.

RENAMES

2 The DATA DIVISION statement in which the RENAMES clause is used is always assigned the level number ________________ (number).

66

REFERENCE MODIFICATION

In the revised language, it is possible to reference a portion of a data field without using REDEFINES or group items in the data description.

The general format is:

```
data-name-1 (leftmost-character-position: [length])
```

Suppose that we have a field A, containing a 9-digit social security number and we want to access the last four digits and store them in a field called B.

```
01  A  PIC  9(9).
01  B  PIC  9(4).
```

We can use the reference modification feature and write:

```
MOVE A (6:4) TO B.
```

The 6 inside the parentheses specifies the starting byte. The colon is a required separator and the 4 specifies the number of bytes being referenced.

Let us consider how we might do the above tests using the 1974 version of COBOL. There are two alternative ways that could be used.

Alternative 1:	*Alternative 2:*
01 A.	01 A PIC 9(9).
02 A1 PIC X(5).	01 A1 REDEFINES A.
02 A2 PIC X(4).	02 A2 PIC X(5).
	02 A3 PIC X(4).
MOVE A2 TO B.	MOVE A3 TO B.

In the general format the "length" item is optional. Its absence implies a reference to the remainder of the field. Thus

```
MOVE A (6:) TO B
```

means to move to B the data stored in A starting with the sixth byte up to the end of the field.

In essence, the reference modification concept is an indirect definition of a field. As such it reduces the effort that is required to write DATA DIVISION entries. On the other hand it allows the programmer to define fields "on the fly," so to speak, and is counter to the documentation spirit of COBOL. Thus, we

advise using it with care. Still, let us consider an example that illustrates the time-saving effect of the reference modification feature. In Chapter 7, Figure 7-9, the following field was defined:

```
01 REPORT-LINE.
   02 FILLER                  PIC X(10) VALUE SPACES.
   02 SALESMAN-NAME-OUT       PIC X(15).
   02 FILLER                  PIC X(2)  VALUE SPACES.
   02 PRODUCT-NUMBER-OUT      PIC 999.
   02 FILLER                  PIC X(11) VALUE SPACES.
   02 SALES-AMOUNT-OUT        PIC Z,ZZ9.99.
   02 FILLER                  PIC X(8)  VALUE SPACES.
   02 TOTAL-SALES-OUT         PIC $$$,$$9.99.
```

The above field definitions were sufficient for a report containing a salesman heading, product heading, and detail lines. However, for purposes of footings we had to create the definitions shown as SALESMAN-FOOTING, PRODUCT-FOOTING and REPORT-FOOTING in Figure 7-9. Now let us consider the use of reference modification. For a salesman footing condition we could have written the following in the 050-PRINT-SALESMAN-FOOTING paragraph of Figure 7-9:

```
MOVE SPACES TO REPORT-LINE
MOVE '**TOTAL FOR SALESMAN ' TO REPORT-LINE (13:24)
MOVE PREVIOUS-SALESMAN-NAME TO REPORT-LINE (37:15)
MOVE SALESMAN-TOTAL-SALES TO TOTAL-SALES-OUT.
```

The two middle MOVE statements above, which contain the reference modification feature, eliminate the need to define the 01 SALESMAN-FOOTING group item. In a similar manner we could eliminate the need for the PRODUCT-FOOTING and REPORT-FOOTING descriptions in the DATA DIVISION in Figure 7-9.

The above example illustrates some of the positive aspects of reference modification. We used reference modification to simplify the code needed to position header-type data into a working storage field. Such a use does not complicate program logic and does not degrade the understandability of the program. Therefore it is a useful application of reference modification.

Review

1 The feature included with the revised language that permits selective reference to a portion of a data item is the ________________ modification feature.

reference

2 For example, if the first four characters from an eight-character field B are to be moved to C, we could write: MOVE B(________________) TO C.

1:4

3 If the last four characters from an eight-character field B are to be moved to C, we could write: MOVE B (________________) TO C.

5:4 (*or, more simply,* 5:)

4 Because the reference modification feature permits an indirect definition of a field, its use could violate the clear-________________ spirit of COBOL.

documentation

WRITE WITH THE LINAGE CLAUSE

The following format represents an expanded form of the WRITE verb:

```
WRITE record-name                          [FROM identifier-1]
 [                          { {identifier-2}          [LINE ] }  ]
 [ {BEFORE}                 { {integer     }          [LINES] }  ]
 [ {AFTER }   ADVANCING     {                                 }  ]
 [                          { {mnemonic-name}                 }  ]
 [                          { {PAGE         }                 }  ]
 [ AT {END-OF-PAGE}  imperative-statement-1 ]
 [    {EOP        }                         ]
 [ NOT AT {END-OF-PAGE}  imperative-statement-2 ]
 [        {EOP        }                         ]
 [END-WRITE]
```

The AT END-OF-PAGE clause serves as a conditional statement. When specified, a check is made to determine if the END-OF-PAGE (abbreviated EOP) condition is met. If it is, the imperative statement is executed. The NOT AT END-OF-PAGE clause, available with the new version of COBOL, also makes it possible to test for the NOT EOP condition.

The END-OF-PAGE or NOT END-OF-PAGE condition is defined by means of the LINAGE clause in the DATA DIVISION, which has the following format:

```
LINAGE is {data-name-a} LINES  [ WITH FOOTING AT {data-name-2} ]
          {integer-1  }        [                 {integer-2  } ]

[ LINES AT TOP {data-name-3} ]  [ LINES AT BOTTOM {data-name-4} ]
[              {integer-3  } ]  [                 {integer-4  } ]
```

Let us consider an example. We want to produce a report with the following format:

LINE NUMBER	CONTENTS
1–4	Not used
6	The page header
7–56	The body of the report
57–59	The page totals
60–66	Not used

We could proceed as follows:

```
              DATA DIVISION.
                   .
                   .
                   .
              FD   PRINT-FILE LABEL RECORD OMITTED
                             DATA RECORD IS PRINT-REC
                             LINAGE IS 54 LINES
                                WITH FOOTING AT 51
                                LINES AT TOP 5
                                LINES AT BOTTOM 7.
```

The page will consist of 66 lines, which is the sum of the values referenced in each phrase except for the FOOTING phrase. Five lines are unused at the top, and seven at the bottom.

In the PROCEDURE DIVISION, the statement:

```
WRITE PRINT-REC FROM TOP-HEADER AFTER ADVANCING PAGE
```

will cause printing of the header on line 6 because now PAGE is associated with line 6, since LINES AT TOP 5 specifies that five lines be left blank at the top of the page. (TOP-HEADER in this example is assumed to contain the desired header.)

Now consider these statements:

```
WRITE PRINT-REC FROM BODY-OF-REPORT-LINE
    AFTER ADVANCING 1 LINE
    AT END-OF-PAGE PERFORM TOTALS.
TOTALS.
    WRITE PRINT-REC FROM TOTALS-LINE
        AFTER ADVANCING 3 LINES
    WRITE PRINT-REC FROM TOP-HEADER
        AFTER ADVANCING PAGE.
```

We will keep printing data from BODY-OF-REPORT-LINE until we have reached line 56, (51 + 5), which is defined as the footing: WITH FOOTING AT 51. At this point, the END-OF-PAGE condition will hold and we will PERFORM TOTALS, in which we print data on line 59 (triple spacing) and then skip to the next page (line 6 of the next page) to print the page header, TOP-HEADER.

A special counter is used whenever LINAGE is specified. It is called LINAGE-COUNTER, a COBOL reserved word. It is set to 1 when a print file is opened or when an ADVANCING PAGE is encountered. Afterward, the counter is automatically incremented the appropriate number of lines implied in each WRITE statement. When LINAGE-COUNTER is equal to the value of the FOOTING phrase, then an END-OF-PAGE condition occurs. The LINAGE-COUNTER may not be modified by the program, but it may be accessed. Thus, it is legitimate to write: IF LINAGE-COUNTER = 25 PERFORM MID-PAGE ROUTINE.

Review

1 When an END-OF-PAGE (EOP) condition is specified in conjunction with a WRITE statement, then the description of the number of lines and their use has to be defined in a(n) ______________ clause in the DATA DIVISION.

LINAGE

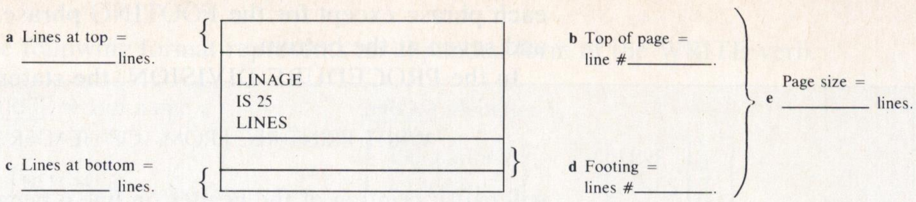
2 The special counter that is a reserved word and is used implicitly whenever the LINAGE option is specified is the ______________.

LINAGE-COUNTER

3 Assume these entries:

```
LINAGE IS 25 LINES
WITH FOOTING AT 21
LINES AT TOP 2
LINES AT BOTTOM 3
```

Fill in the missing members:



a 2
b 3
c 3
d 23 − 27
e 25 + 2 + 3 = 30

THE MOVE CORRESPONDING OPTION

Earlier in this chapter we described the concept of qualification. Defined briefly, this concept allows the programmer to use the same subordinate data-name in more than one place in the program, thus allowing nonunique data-names. The CORRESPONDING option, available for use with MOVE and with the arithmetic verbs, simplifies the program in cases in which the same operation is to be performed on one or several pairs of elementary, nonunique data-names. Let us take an example. Suppose we have the following two records:

```
01  PAY-RECORD.
    02  GROSS     PIC 9999V99.
    02  NET       PIC 9999V99.
    02  TAXES     PIC 999V99.
01  EDITED-RECORD.
    02  GROSS     PIC ZZZ9.99.
    02  TAXES     PIC ZZ9.99.
    02  NET       PIC ZZZ9.99.
```

If we want to move PAY-RECORD to EDITED-RECORD, we *cannot* do it in one statement. Writing MOVE PAY-RECORD to EDITED-RECORD would be incorrect, because the order of the fields NET and TAXES is not the same in the two records and there are edited fields in the receiving record. Of course, the move could be accomplished by a separate MOVE statement for each of the three fields; however, the same result can be accomplished more easily by use of the CORRESPONDING option:

```
MOVE CORRESPONDING PAY-RECORD TO EDITED-RECORD.
```

The general format associated with the use of the CORRESPONDING option is:

```
MOVE {CORRESPONDING} identifier-1 TO identifier-2
     {CORR         }
```

CORR is the abbreviated form of the option. Unlike the situation in the previous example, the two data-names may contain only some items that correspond, as in the following example:

```
02  INSPECTION.
    03  TOTAL-QUANTITY . . .
    03  REJECTED . . .
```

```
        03  ACCEPTED . . .
        03  QUALITY-RATIO . . .

    01  QUALITY-REPORT.
        02  TOTAL-QUANTITY . . .
        02  QUALITY-RATIO . . .
```

Executing the statement MOVE CORR INSPECTION TO QUALITY-REPORT will result in the two items or fields, TOTAL-QUANTITY and QUALITY-RATIO, being moved.

In order for the CORRESPONDING option to be used, there must be pairs of items having the same name in two group items, and at least one of the items in each pair must be elementary. Another rule to remember is that any items that are subordinate to identifier-1 and identifier-2 and have RENAMES, REDEFINES, or OCCURS clauses are ignored in the move. Therefore, we cannot use the MOVE CORRESPONDING option to move a table of values, for example. However, the identifier-1 and identifier-2 items themselves may have REDEFINES or OCCURS clauses or may be subordinate to data items with such clauses. (These clauses are discussed in later chapters.)

The CORRESPONDING option also is available with ADD and SUBTRACT, as can be observed in Appendix B: "Complete ANS COBOL Language Formats." In general, the option should be avoided or used sparingly with respect to both MOVE and the arithmetic verbs. Use of the CORRESPONDING option may result in errors when programs are modified subsequently.

Review

1 The abbreviated form of the CORRESPONDING option is ______. Use of this option in conjunction with the MOVE instruction results in transfer of only the ______ items contained in two records.

CORR; *common (or corresponding)*

2 When the MOVE CORRESPONDING option is used, an item will be moved if at least one of the items in each pair is at the ______ level and only if the receiving group item has an item with the same [storage capacity / name].

elementary; name

3 The MOVE CORRESPONDING option can be used to move elementary items [including / but not including] tables of values at the elementary level.

but not including

THE STRING AND UNSTRING VERBS

The STRING and UNSTRING verbs are designed to facilitate transfer of data from several sources into one destination and from one source to many destinations, respectively. In effect, use of these verbs allows one statement to be used in lieu of multiple uses of the MOVE verb and, possibly, in lieu of some DATA DIVISION entries.

We begin with two examples that illustrate uses of the STRING verb.

Suppose that EDIT-SOC-SEC contains a social security number, including hyphens after the third and fifth digits, as, for instance, '123-45-6789'. We wish

to move the social security number to SOC-SEC while also removing the hyphens. The following data description entries are given:

```
01  SOC-SEC            PIC X(9).
01  EDIT-SOC-SEC.
    02  PART-1         PIC 999.
    02  FILLER         PIC X   VALUE '-'.
    02  PART-2         PIC 99.
    02  FILLER         PIC X   VALUE '-'.
    02  PART-3         PIC 9999.
```

We now use the STRING statement:

```
STRING  PART-1 DELIMITED BY SIZE
        PART-2 DELIMITED BY SIZE
        PART-3 DELIMITED BY SIZE
  INTO  SOC-SEC.
```

The STRING specifies moving the three fields, PART-1, PART-2, and PART-3, and positioning them adjacent to each other. The transfer of data can be thought of as taking place character by character. Thus, the data in PART-1 would be transferred into the first three positions of SOC-SEC, the data in PART-2 would be transferred into the next two positions of SOC-SEC, and so on. The DELIMITED BY SIZE clause specifies that the transfer of data from the associated field will stop at (be delimited by) the point when as many characters have been transferred as the size of the source field. The next example illustrates the availability of other alternatives.

Assume that we want to print a report that lists a company name in columns 5–20, a city name starting with column 26, 1 blank space, and then the ZIP code. The source of data is VENDOR-RECORD:

```
01  VENDOR-RECORD.
    02  COMPANY-NAME        PIC X(15).
    02  STREET              PIC X(40).
    02  CITY-STATE          PIC X(20).
    02  ZIP                 PIC 9(5).
```

The data in CITY-STATE are recorded so that the city-name is followed by a comma, a space, and then the state code, e.g., LOS ANGELES, CA.

The output record is described as:

```
01  OUTPUT-REC   PIC X(132).
```

We use the STRING verb as follows:

```
MOVE SPACES TO OUTPUT-REC
MOVE 5 TO STARTING-PLACE.
STRING COMPANY-NAME DELIMITED BY SIZE
      '     '             DELIMITED BY SIZE
      CITY-STATE          DELIMITED BY ','
      SPACE, ZIP          DELIMITED BY SIZE
      INTO OUTPUT-REC
      WITH POINTER STARTING-PLACE.
```

The first MOVE statement clears the output record of any previous contents. The second MOVE sets STARTING-PLACE to a value of 5 so that the beginning of data transfer into OUTPUT-REC will begin in column 5 (WITH POINTER STARTING-PLACE). Of course, STARTING-PLACE is an arbitrary name chosen by the programmer; but it must be an integer field for obvious reasons.

The STRING statement specifies that, in effect, five fields will be transferred:

```
STRING  {identifier-1}  {, identifier-2}  . . . DELIMITED BY {identifier-3}
        {literal-1   }  {, literal-2   }                     {literal-3   }
                                                             {SIZE        }

 { {identifier-4}  {, identifier-5}  . . .  DELIMITED BY  {Identifier-6} }
 {,{literal-4   }  {, literal-5   }                       {literal-6   } } . . .
 {                                                        {SIZE        } }

INTO identifier-7 [WITH POINTER identifier-8]
[ON OVERFLOW imperative-statement-1]
[NOT ON OVERFLOW imperative-statement-2]
[END-STRING]
```

FIGURE 8-2
GENERAL FORMAT FOR THE STRING VERB.

COMPANY-NAME, the nonnumeric literal '   ', CITY-STATE, the figurative constant SPACE, and ZIP. Thus, starting with column 5 of OUTPUT-REC, the entire (DELIMITED BY SIZE) COMPANY-NAME is transferred and it is followed by the 5-blank nonnumeric constant. The next data item to be transferred comes from CITY-STATE; this data item is transferred character by character until a comma is encountered (DELIMITED BY ','). One blank follows (SPACE) and then the ZIP code. It should be pointed out that use of figurative constants, such as SPACE, ZEROS, or the like, always means one occurrence of the implied character. Thus, we would obtain one blank even if we had used SPACES instead of SPACE.

The general format for the STRING verb is presented in Figure 8-2.

Our two examples have illustrated all but the OVERFLOW option. If the data specified to be transferred are greater than the size of the receiving item (identifier-7) during execution of a STRING statement, then the imperative statement of the OVERFLOW clause is executed. If the optional OVERFLOW is not used and the overflow condition arises, then the STRING operation is discontinued and the next program statement is executed. During execution, identifier-8, if used, is incremented by 1 as each character is transferred. It is the value of this identifier that is checked in determining an overflow condition. If identifier-8 is not used, an implied counter is used to fulfill the same function. The NOT ON OVERFLOW and the END-STRING options apply to the revised language only.

The UNSTRING verb, as its name implies, acts in the reverse direction of the STRING verb. We present two examples to illustrate use of this verb.

Suppose that data are recorded in free form (without predefined fields) as follows:

```
TED S BROWN,4,15,3.52
TINA LORI CHRISTIANSON,1,12,2.50
```

As we can see, name fields are separated by 1 or more blank spaces; then commas separate the remaining three fields. We would like to move these data fields to a fixed-format record:

```
01  STUDENT-RECORD.
    02  FIRST-NAME        PIC X(15).
    02  MIDDLE-NAME       PIC X(15).
    02  LAST-NAME         PIC X(20).
    02  CLASSIFICATION    PIC 9.
    02  CREDIT-LOAD       PIC 99.
    02  GPA               PIC XXXX.
```

Assuming that the source data are in

```
01  FREE-FORM-RECORD   PIC X(57).
```

we can write:

```
UNSTRING  FREE-FORM-RECORD
          DELIMITED BY ALL SPACES OR ','
    INTO  FIRST-NAME
          MIDDLE-NAME
          LAST-NAME
          CLASSIFICATION
          CREDIT-LOAD
          GPA.
```

The DELIMITED clause specifies that fields in the source record are separated by 1 or more blank spaces (ALL SPACES), or single commas (OR ','). In essence, the source record is scanned character by character from left to right. When a blank or a comma appears, it is assumed that a new field begins. The delimiters in this case are blanks or commas, and they are not included in the data transfer, although the UNSTRING statement does include an option allowing the transfer of delimiters themselves.

Consider now a second example that expands on UNSTRING and illustrates combined use of STRING and UNSTRING.

Data records contain numbers in columns 1–6, followed by a name and a header separated from each other by a dollar sign. As in the previous example, a delimiter such as a dollar sign can be used to allow recording of data without adherence to predefined field positions. When data length is highly variable, such free-form data can save a lot of space. Two sample records are as follows:

```
349687INTERNATIONAL TOOLS, INC. $ BALANCE SHEET $
135002ACME CORP. $ INCOME STATEMENT $
```

We are interested in printing the company name, centered at column 40 on the top of a new page, followed by the name of the report on the third line, also centered at column 40. Solution of the problem involves separating the two fields, determining their size, and, on the basis of their size, centering the data with respect to column 40. We also assume that we wish to ascertain that there are indeed two fields available in the relevant part of the source record. First, we define some data fields:

```
01  FREE-FORM-RECORD    PIC X(46).
01  FIRST-LINE          PIC X(20).
01  SECOND-LINE         PIC X(20).
01  LENGTH-1            PIC 99.
01  LENGTH-2            PIC 99.
01  STARTING-POINT      PIC 99.
01  NO-OF-FIELDS        PIC 9.
01  OUTPUT-RECORD       PIC X(132).
```

Figure 8-3 presents a program segment written to accomplish the desired task. The NO-OF-FIELDS item is used to count the number of fields transferred. Notice its use in the TALLYING clause in the UNSTRING statement. The value 7 is moved to STARTING-POINT because the first 6 columns of FREE-FORM-RECORD contain data that we wish to ignore. Notice the clause WITH POINTER STARTING-POINT. Using the $ as a delimiter, we transfer data from the source record into two fields, FIRST-LINE and SECOND-LINE. In the process, we obtain a count of the characters moved into each receiving field in LENGTH-1 and LENGTH-2, respectively. The COUNT option provides this length-count.

```
MOVE ZERO TO NO-OF-FIELDS
MOVE 7 TO STARTING-POINT
UNSTRING FREE-FORM-RECORD DELIMITED BY '$'
  INTO FIRST-LINE
      COUNT IN LENGTH-1
    SECOND-LINE
      COUNT IN LENGTH-2
  WITH POINTER STARTING-POINT
  TALLYING IN NO-OF-FIELDS
  ON OVERFLOW
    PERFORM ERROR-ROUTINE-1.
IF NO-OF-FIELDS NOT = 2
  PERFORM ERROR-ROUTINE-2
ELSE
  PERFORM PRINT-HEADERS.
.
.
.
PRINT-HEADERS.
  COMPUTE LENGTH-1 = 40 - (LENGTH-1 / 2)
  COMPUTE LENGTH-2 = 40 - (LENGTH-2 / 2)
  MOVE SPACES TO OUTPUT-RECORD
  STRING FIRST-LINE DELIMITED BY SIZE
    INTO OUTPUT-RECORD
    WITH POINTER LENGTH-1
  WRITE OUTPUT-RECORD AFTER ADVANCING PAGE
  MOVE SPACES TO OUTPUT RECORD
  STRING SECOND-LINE DELIMITED BY SIZE
    INTO OUTPUT-RECORD
    WITH POINTER LENGTH-2
  WRITE OUTPUT-RECORD AFTER ADVANCING 2 LINES.
```

FIGURE 8-3
EXAMPLE PROGRAM INVOLVING THE USE OF UNSTRING AND STRING.

Finally, use of the OVERFLOW specifies execution of PERFORM ERROR-ROUTINE-1 if the data being transferred exceed the size of the receiving field. This could happen in our example if the delimiting dollar sign was missing, or if one field was longer than 20 characters—the size specified for FIRST-LINE and SECOND-LINE.

After the UNSTRING statement, we check to see that we indeed had two fields transferred; if not, we PERFORM ERROR-ROUTINE-2.

The PRINT-HEADERS paragraph computes the starting point of each line to the left of column 40. We divide the length of the field involved by 2 and we subtract this amount from 40. We then use the STRING verb to move the data, using LENGTH-1 as the pointer. Actually, it is the availability of the POINTER option in the STRING verb that makes it capable of achieving what the MOVE verb could not accomplish in this case. After the transfer of the data, we print the record and repeat the process for the next line of printed output. The general format for the UNSTRING verb is presented in Figure 8-4.

We have illustrated all the options except the DELIMITER IN clause. When used, the clause specifies the identifier to be used to receive the delimiter(s). This option is used if we wish to capture the delimiters themselves for further testing or processing.

```
UNSTRING identifier-1  { DELIMITED BY [ALL] {identifier-2} {OR
                                             {literal-1   }

[ALL] {identifier-3}} . . . } INTO identifier-4  [, DELIMITER IN identifier-5]
      {literal-2   }}

[, COUNT IN identifier-6]  [, identifier-7  [, DELIMITER IN identifier-8]
[, COUNT IN identifier-9] ] . . . [WITH POINTER identifier-10]
[TALLYING IN identifier-11]  [ON OVERFLOW imperative-statement-1]
[NOT ON OVERFLOW imperative-statement-2]
[END-UNSTRING]
```

FIGURE 8-4
GENERAL FORMAT FOR THE UNSTRING VERB.

Review

1 The verb that is used to transfer data from several sources to one destination is the ________ verb.

STRING

2 The verb that is used to transfer data from one source to many destinations is the ________ verb.

UNSTRING

3 When the DELIMITED BY SIZE clause is used in conjunction with the STRING verb, transfer of data from the sending field stops when the number of characters which have been transferred equals the size of the [sending / receiving] field.

sending

4 If an OVERFLOW clause is not used in conjunction with a STRING verb and an overflow condition occurs, then [the STRING operation / program execution] is terminated.

the STRING *operation*

5 Used in conjunction with the UNSTRING verb, the DELIMITED BY clause specifies the basis used to signal the beginning of a new [sending / receiving] field.

sending

6 The clause that is used if delimiters themselves, such as commas or spaces, are to be transferred to receiving fields during the UNSTRING operation is called the ________ clause.

DELIMITER IN

THE INSPECT VERB

At times we need to access and manipulate individual characters in a field. One very common use is to edit input data, such as replacing leading blanks by zeros. COBOL provides the INSPECT verb to accomplish such character manipulations. This verb replaces the EXAMINE verb, which served a similar but more limited purpose in previous versions of the language.

The INSPECT verb is powerful but a bit complicated. Three formats are available, and these are presented in the format specifications in Appendix B. Discussion of the complete set of options would exceed the intended scope of this text. We present some examples to illustrate the basic options.

EXAMPLE 1: Suppose we want to replace all *leading* blanks by leading zeros in a field called TEST. We could write:

```
INSPECT TEST REPLACING LEADING ' ' BY '0'.
```

EXAMPLE 2: Suppose we want to replace *all* blanks by zeros in a field called TEST. We could write:

```
INSPECT TEST REPLACING ALL ' ' BY '0'.
```

EXAMPLE 3: If we want to replace the first zero by a +, we write:

```
INSPECT TEST REPLACING FIRST '0' BY '+'.
```

EXAMPLE 4: Suppose we want to ask the question: How many dollar signs are in TEST? We write:

```
INSPECT TEST TALLYING COUNT-A FOR ALL '$'.
```

After this instruction is executed, the numeric field COUNT-A will contain a value equal to the number of $ in TEST. (COUNT-A would have been defined in the DATA DIVISION.)

EXAMPLE 5: Suppose we want to ask the question: How many zero characters are there to the left of the decimal point and how many zeros are there to the right of the decimal point? We write:

```
INSPECT TEST TALLYING COUNT-A FOR ALL '0' BEFORE INITIAL '.'
                      COUNT-B FOR ALL '0' AFTER '.'.
```

This instruction would result in COUNT-A containing the number of zeros before the decimal point and COUNT-B containing the number of zeros after the decimal point.

EXAMPLE 6: We want to count the number of dollar signs in TEST and replace all dollar signs after the first one by asterisks. We write:

```
INSPECT TEST TALLYING COUNT-A FOR ALL '$'
        REPLACING ALL '$' BY '*' AFTER INITIAL '$'.
```

EXAMPLE 7: We want to ask the question: Assuming that TEST contains a name left-justified, how long is the name? (Unused positions are blank.) We write:

```
INSPECT TEST TALLYING COUNT-A FOR CHARACTERS BEFORE INITIAL ' '.
```

EXAMPLE 8: We want to convert all uppercase letters to lowercase. We could write:

```
INSPECT TEST REPLACING ALL 'A' BY 'a'
                           'B' BY 'b'
                            .
                            .
                            .
                           'Z' BY 'z'.
```

In some instances the compiler may limit the number of literal pairs used in one INSPECT statement, so that for the example we may need to write two INSPECT statements, say, one for replacing the letters A through M and another one for the letters N through Z.

In the revised language there is an easier way to do the above tests by using a new option, CONVERTING:

```
INSPECT TEST CONVERTING 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
                     TO 'abcdefghijklmnopqrstuvwxyz'
```

The above statement specifies to change any "A" to "a," and so on, exactly as the REPLACING ALL statement does at the beginning of this example.

EXAMPLE 9: An integer field ABC of six positions may have leading blanks and a sign. We want to move the correct numeric value represented in ABC to S-ABC whose PIC is S9(6). We could write:

```
MOVE ZERO TO PLUS-SIGN, MINUS-SIGN
INSPECT ABC
    TALLYING
        PLUS-SIGN FOR ALL '+'
        MINUS-SIGN FOR ALL '-'
    REPLACING
        LEADING SPACES BY ZERO
        FIRST '+' BY ZERO
        FIRST '-' BY ZERO.
IF ABC IS NOT NUMERIC
    PERFORM INCORRECT-DATA
ELSE
    MOVE ABC TO S-ABC
    IF MINUS-SIGN NOT = ZERO
        MULTIPLY -1 BY S-ABC.
```

We initialize two counters, PLUS-SIGN and MINUS-SIGN, to zero and we use them for TALLYING the occurrence of + and −, respectively. Notice, however, that we use REPLACING . . . FIRST. In the unlikely event that more than one like sign is present, the statement IF ABC IS NOT NUMERIC would sense the presence of the unconverted extra sign(s). If the field is unsigned or positive, MOVE ABC to S-ABC is sufficient. If the field is negative, however, then MINUS-SIGN would have a nonzero value and we multiply S-ABC by −1 to attain the proper sign.

Review

1 Individual characters in a field can be accessed and possibly changed by use of the ________________ verb.

INSPECT

2 Use of the TALLYING option in conjunction with the INSPECT verb makes it possible to ________________ designated characters.

count

3 Use of the REPLACING option in conjunction with the INSPECT verb makes it possible to ________________ designated characters.

change

4 In the revised language, a more efficient way than use of the REPLACING option by which a *series* of designated characters can be changed is use of the new option, ____________.

CONVERTING

THE INITIALIZE VERB

The INITIALIZE statement provides for a convenient way to move data to selected fields. The most common use of this verb is for setting numeric fields to zero and nonnumeric fields to spaces.

Figure 8-5 presents the general format and some examples of using the verb. When used without the REPLACING option, spaces are moved to alphabetic, alphanumeric, and alphanumeric-edited data items, while zeros are moved to numeric and numeric edited items. When identifier-1 is a group item, as in the first INITIALIZE A example, it is as if we wrote a series of elementary MOVE statements to move either spaces or zeros, depending on the receiving data item category.

When the REPLACING option is used, only items matching the specified category are initialized. Thus, the third example in Figure 8-5*b* illustrates that only D is affected by the INITIALIZED statement.

```
INITIALIZE   {identifier-1} . . .

[              { ALPHABETIC              }                               ]
[              { ALPHANUMERIC            }                               ]
[ REPLACING    { NUMERIC                 }  DATA BY  {identifier-2}  . . .]
[              { ALPHANUMERIC-EDITED     }           {literal     }      ]
[              { NUMERIC EDITED          }                               ]
```

(*a*) General Format

DATA DIVISION SPECIFICATIONS

```
01 A.
     02  B    PIC   A(2).
     02  C    PIC   X(2).
     02  D    PIC   9V9.
     02  E    PIC   X/X.
     02  F    PIC   $9.99.
     02  G    PIC   X(2).
01  M   PIC   9V9   VALUE 1.2.
01  N   PIC   XX     VALUE '12'.
```

INITIALIZE Statement Examples	Equivalent MOVE Statement(s)
INITIALIZE B	MOVE SPACES TO B
INITIALIZE A	MOVE SPACES TO B, C, E, G MOVE ZERO TO D, F
INITIALIZE A REPLACING NUMERIC BY ZERO	MOVE ZERO TO D
INITIALIZE A REPLACING ALPHABETIC DATA BY 'NO' ALPHANUMERIC DATA BY N NUMERIC DATA BY M NUMERIC-EDITED DATA BY 9.1 ALPHANUMERIC-EDITED DATA BY N	 MOVE 'NO' TO B MOVE N TO C, G MOVE M TO D MOVE 9.1 TO F MOVE N TO E

(*b*) Sample Uses

FIGURE 8-5

FORMAT AND SAMPLE USES OF INITIALIZE.

Overall, we can observe that the INITIALIZE serves as a convenient alternative to the VALUE clause in the DATA DIVISION and the use of MOVE in the PROCEDURE DIVISION.

Review

1 In effect, the INITIALIZE verb available with the revised language represents a more efficient alternative to the use of the ________________ verb.

MOVE

2 The most common use of this verb is for setting numeric fields to ________ and nonnumeric fields to ____________.

zero; spaces

THE COPY VERB

COBOL provides for a library facility. By ''library'' we mean a collection of COBOL source program elements recorded on tape or disk and accessible by reference to *text-names*. A well-planned and maintained library can reduce the time needed to write routines common to several programs, and it can serve to standardize such common routines.

For example, suppose we include the following staement in the DATA DIVISION:

```
    01  INVENTORY-RECORD COPY MATERIAL-RECORD.
```

We obtain the following source program listing at compilation time. The C characters on the left margin indicate that these entries were copied.

```
      01  INVENTORY-RECORD COPY MATERIAL-RECORD.
C         02  PART-NUMBER       PICTURE X(9).
C         02  PART-NAME         PICTURE X(15).
C         02  STOCK-QUANTITY    PICTURE 9(6).
```

We can use the copied items in the PROCEDURE DIVISION as if we had written their description in the DATA DIVISION of the current program. Notice that the record-name used in this program example, INVENTORY-RECORD, is not the same as the library-name for that record. Use of the same name is optional. It is good procedure, of course, to use the same name, so that comparison of different programs and communication between programmers are facilitated. The library simplifies program writing by standardizing records in different files so that programs can be maintained and revised easily.

Building the library commonly is done outside the COBOL language by means of JCL (Job Control Language) statements. Use of a library is a very *local* practice, and we direct the readers to their own computer system for details.

Review

1 The verb that makes it possible to reference a precoded program segment from a library of such program segments is ________________.

COPY

2 When the COPY verb is used to obtain a record description from a library, the record-name used in the program and in the library [must / need not] be the same.

need not

EXERCISES

8.1 A large company has three subsidiaries that formerly were independent companies. Each subsidiary has a different product code, as follows:

Subsidiary 1 uses a 4-character product code.
Subsidiary 2 uses a 6-character product code.
Subsidiary 3 uses a 7-character product code.

Apart from the difference in length, the product code can be used in a program that does some processing for all subsidiaries. The data in the program are contained in a field called PRODUCT-CODE, which has a size of seven characters.

a Write a DATA DIVISION description of PRODUCT-CODE so that PRODUCT-CODE can be tested to determine which subsidiary's product code it contains and so that the correct size product code can be MOVED. Assume that the data in PRODUCT-CODE is left-justified, with blanks filling the unused space. (*Hint:* REDEFINES or RENAMES may be useful.)

b Write PROCEDURE DIVISION statements to accomplish the processing objectives described in part **a**.

8.2 Assume the following DATA DIVISION entries:

```
01 A.
   02 X.
      03 Y.
   02 W.
      03 Y.
01 B.
   02 X.
      03 Y.
   02 W.
      03 Y.
```

Write PROCEDURE division statement(s) to move the last elementary item in A to the first elementary item in B.

8.3 A record file contains name and address data for college students and their parents or guardians. The file is arranged so that for each student there are two records. The first one contains the name and address of the student, and the second one contains the name and address of parent or guardian. The record formats are as follows:

STUDENT		PARENT	
FIELD	COLUMNS	FIELD	COLUMNS
Student number	1–9	Student number	1–9
Student name	10–30	Parent name	10–30
Street	31–60	Street	31–60
City	61–79	City	61–79
Record code = 1	80	Record code = 2	80

a Write ENVIRONMENT and DATA DIVISION file and record entries to describe this file.

b Write PROCEDURE DIVISION statements to read two consecutive records, testing to ascertain that they are student and parent records, respectively. When the first record is read, if it is a student record, it is

stored in INPUT-WORK-AREA. If the first record is not a student record or if the second record is not a parent record for the same student, the program performs a paragraph called ERROR-ROUTINE. If the records are correct, the program performs a paragraph called PROCESS.

8.4 In the following diagram there is a record called BIGFIELD. The numbers running from 1 to 13 indicate respective character positions. Thus, the record consists of 13 character positions. We want to be able to reference the following positions while also preserving the current structure of the record. Indicate how to accomplish this objective.

<table>
<tr><td colspan="13">BIGFIELD</td></tr>
<tr><td colspan="7">GROUP-A</td><td colspan="6">GROUP-B</td></tr>
<tr><td colspan="2">AA</td><td colspan="3">AB</td><td colspan="2">AC</td><td colspan="2">BA</td><td colspan="2">BB</td><td colspan="2">BC</td></tr>
<tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td></tr>
</table>

a Reference 1, 2, 3, 4 by one name
b Reference 5, 6, 7 by one name
c Reference 8, 9, 10 by one name
d Reference 11, 12, 13 by one name
e Reference 1, 2, 3, 4, 5, 6, 7 by one name
f Reference 8, 9, 10, 11, 12, 13 by one name
g Reference 3, 4, 5, 6, 7, 8, 9, 10 by one name

8.5 A file contains customer names in free form. The last name is written first, followed by a comma, followed by the first name or initial, followed by a comma, and finally followed by the middle name or initial. Write a program to read data from that file and print the data in the following fixed format:

Last name: columns 11–30
First name: columns 32–45
Middle name: columns 47–60

Print the data under suitable headings and use the following test data:

SIMON, AL, GEORGE
GARCIA, P., K.
LONGNAME, VALERIE, GEORGIA

8.6 Write a COBOL program to compute depreciation schedules, using the sum-of-the-digits method of depreciation. The sum-of-the-digits-method works as follows: Suppose that you have an asset of original value $1,000.00 to be depreciated over 3 years, using the sum-of-the-digits method. The following table shows the nature of the calculations involved:

YEAR	DEPRECIATION RATE	DEPRECIATION
1	$\frac{3}{1+2+3} = \frac{3}{6}$	$1{,}000 \times \frac{3}{6} = 500.00$
2	$\frac{2}{1+2+3} = \frac{2}{6}$	$1{,}000 \times \frac{2}{6} = 333.33$
3	$\frac{1}{1+2+3} = \frac{1}{6}$	$1{,}000 \times \frac{1}{6} = 166.67$

Notice that the depreciation rate varies from year to year, but that the rate is applied to the same (original) asset dollar value. The rate consists of a

denominator that is the sum of the digits from 1 up to the number of years over which the asset is to be depreciated. To test your understanding of the concept, compute the denominator value for five years. The answer is 15. The numerator of the depreciation rate varies from the number of years to 1 in steps of 1. Thus, for a five-year depreciation schedule the numerator values are: 5, 4, 3, 2, 1.

Input: This is in the form of a set of records with the following data format:

COLUMNS	CONTENT
1–5	Asset number
6–14	Original asset value in dollars and cents
15–16	Blank
17–18	Number of years over which the asset is to be depreciated
19–80	Blank

Output: This should be on a new page for each asset and should have approximately the following format:

ASSET 12345	ORIGINAL VALUE: $1,000.00		
		ACCUMULATED	
YEAR	DEPRECIATION	DEPRECIATION	BOOK VALUE
1	$500.00	$ 500.00	$500.00
2	333.33	833.33	166.67
3	166.67	1,000.00	0.00

Note: In order always to show a final accumulated depreciation equal to the original value, as well as a final book value equal to zero, the depreciation of the last year is to be computed as follows:

$$\text{Last year depreciation} = \text{original value} - \text{accumulated depreciation}$$

Write and run a COBOL program to produce the desired output. Use as test data those shown in the preceding description.

8.7 A file contains data about the inventory of a company. Each inventory item is identified by a unique item number. The file is sorted on item number. There are three types of records in the file. A *balance* record contains the amount in inventory as of the last time the data were processed. A *receipt* record contains the amount of a shipment received. An *issue* record contains the amount sold. For each shipment received and each sale made, a separate record is created. We wish to read the records, process the data, produce an inventory report on the printer, and produce a set of new balance records.

Input is as follows:

COLUMN	CONTENT
1–5	Item number
6–20	Part name (on balance records only)
21	Type code 1 = balance 2 = receipt 3 = issue
22–25	Quantity

Note: The records are sorted on item number. For each item, the balance record *precedes* the receipt and issue records.

Sample input is:

```
01212TRANSFORMER      12350
01212                 26000
01212                 30150
01212                 33050
01212                 31600
01515GEAR TRAIN       11000
01515                 30600
02010METAL PLATE      14000
```

Sample output: The printer output resulting from the previously given sample input is:

ITEM NUMBER	PART NAME	PREVIOUS BALANCE	NEW BALANCE
01212	TRANSFORMER	2350	3550
01515	GEAR TRAIN	1000	400
02010	METAL PLATE	4000	4000

The balance records resulting from the sample input are:

```
01212TRANSFORMER      103550
01515GEAR TRAIN       100400
02010METAL PLATE      104000
```

These balance records serve as input to the next program run, along with the receipt and issue records created between processing runs.

Write and run a COBOL program to accomplish the desired result. Assume all data are correct.

8.8 The data processing objective of this program is to compute and print the monthly schedule of payments resulting from a credit purchase. Given the amount of the credit purchase and the number of monthly payments planned, the amount of the monthly payment is computed as follows:

$$\text{Payment due} = \frac{\text{amount of credit purchase}}{\text{number of payments}} + 0.015 \text{ of unpaid balance}$$

Of course, this formula presumes an interest charge of 0.015 per month on the unpaid balance in the account. For example, suppose that a customer has purchased an item valued at \$1,200.00 and is going to pay for it over a 12-month period. The payment due at the end of the first month is:

$$\text{Payment due} = \frac{\$1{,}200.00}{12} + 0.015\ (\$1{,}200.00)$$

$$= \$100.00 + 0.015\ (\$1{,}200.00) = \$100.00 + \$18.00 = \$118.00$$

Similarly, for the second month the payment is:

$$\text{Payment due} = \frac{\$1{,}200.00}{12} + 0.015\ (\$1{,}100.00)$$

$$= \$100.00 + 0.015\ (\$1{,}100.00) = \$100.00 + \$16.50 = \$116.50$$

The monthly payment consists of a constant element, which is the original amount of the credit purchase divided by the number of monthly payments, and a variable element, which is the monthly interest charge on the unpaid balance. With each monthly payment, the unpaid balance is decreased by the amount of the constant element.

The desired output for this program is illustrated in Figure 8-6. This case involves a credit transaction of $4,291.50. It is assumed that the letterhead is preprinted, but that the name and address, all column headings, and the numeric values will be printed as the result of program execution. Since the column headings are always the same, in practice these headings also would be preprinted on the form in order to conserve computer time. However, in our example we will print the headings in order to illustrate how this can be accomplished through the use of COBOL. For each monthly payment, the numeric information provided is the amount of the monthly interest charge, the total payment due that month, and the unpaid balance remaining after that month. Note also that the spacing of the computer printout is

ABC COMPANY
5000 East Camelback Road
Phoenix, Arizona 85033

ALBERT ROBINSON
3150 NORTH ST.
TEMPE, AZ 85282

Fold here -

SCHEDULE OF PAYMENTS
ORIGINAL AMOUNT $4,291.50

MONTH	INTEREST	TOTAL PAYMENT	UNPAID BALANCE
1	64.37	422.00	3,933.87
2	59.00	416.63	3,576.25
3	53.46	411.27	3,218.62
4	48.27	405.90	2,861.00
5	42.91	400.54	2,503.37
6	37.55	395.18	2,145.75
7	32.18	389.81	1,788.12
8	26.82	384.45	1,430.50
9	21.45	379.08	1,072.87
10	16.09	373.72	714.25
11	10.72	368.35	357.62
12	5.36	362.99	.00

Envelope

Window

ALBERT ROBINSON
3150 NORTH ST.
TEMPE, AZ 85282

FIGURE 8-6
DESIRED FORM OF OUTPUT FOR EXERCISE 8.8.

designed for use with a window envelope, thereby eliminating the need for separate addressing of envelopes.

Program input: This will consist of a file, with one record per customer. The record format and the layout of these data is as follows:

COLUMNS	CONTENTS
1–20	Customer's name
21–45	Number and street
46–70	City, state, and ZIP code
71–78	Amount of credit purchase
79–80	Number of monthly payments

Write a COBOL program to accomplish the task we have described. Use the input for Figure 8-6 as test data for your program, and compare your output with the output in this figure.

9 SEQUENTIAL FILE PROCESSING

- FILE ORGANIZATION
- FILE STORAGE DEVICES
- FILE LABELS
- COBOL INSTRUCTIONS FOR SEQUENTIAL FILES
- THE FILE-CONTROL SPECIFICATION
- THE FILE-DESCRIPTION ENTRY
- THE OPEN AND CLOSE VERBS
- THE READ, WRITE, AND REWRITE VERBS
- SAMPLE PROGRAM TO CREATE A SEQUENTIAL FILE
- SEQUENTIAL FILE UPDATING
- SAMPLE SEQUENTIAL FILE UPDATE PROGRAM
- EXERCISES

FILE ORGANIZATION

The concept of *file organization* refers to the manner in which data records are arranged on a file-storage medium. There are three principal methods of file organization in COBOL: sequential, indexed, and relative.

In a *sequential* file, the records are written in a serial order and are accessed (read) in the same order as written. The serial order need *not* be in any particular sequence, such as according to account number. Files assigned to the printer and magnetic tape drive are always organized as sequential files. On the other hand, files stored on magnetic disk and other direct access storage devices may be sequential, indexed, or relative files.

Although the language does not require it, sequential files on tape or disk most commonly are sorted so that the records are in some logical sequential order. For instance, if we have a customer file we might choose to sort the records on the basis of customer number. Then, if customer numbers are unique, each successive record read from the file should have a higher customer number than the one before. This practice of sorting the records is convenient when processing "batch" jobs, such as payroll. In such jobs, we sort the "transaction" data, such as time cards, in the same order as the employee payroll file. Then we need to go through that file only once, rather than go back and forth looking for employee records in random order.

Indexed file organization means that an index has been created so that records can be located directly without accessing them in sequence. We describe this file organization method in Chapter 13: "Indexed Sequential File Processing."

Relative file organization means that the file is stored in such a way that each record has an identifier by which it can be accessed directly. This method is covered in Chapter 14: "Relative File Processing."

Review

1 The manner in which data records are arranged on a file storage medium is referred to as file ______________.

organization

2 The type of file for which the records are written in a serial order and for which the records must be accessed in the same order as written is the ____ file organization.

sequential

3 The type of file for which the records have been read and stored in a serial order, but for which access can be direct, is the ______________ file organization.

indexed

4 The type of file for which records are both stored and accessed directly according to the value of an identifier is the ______________ file organization.

relative

5 If a file is assigned to a printer or magnetic tape drive, it must be organized as a(n) ______________ file.

sequential

6 A file that can be organized by any of the three methods described in this section is one for which the records are stored on magnetic ______________.

disk

FILE STORAGE DEVICES

While the characteristics of the data processing task dictate which method of file organization is preferred, the storage device that is used determines which methods are possible. As related to file organization, it is useful to distinguish two main categories of storage devices: magnetic tape and mass-storage devices. The latter category refers mainly to magnetic disk, but it also includes other direct access storage devices, such as cartridge systems. As indicated in the preceding section, only the sequential file organization can be used for files stored on magnetic tape, whereas any method of file organization can be used for files stored on direct access storage devices.

The magnetic tape unit is the specialized input-output device used for reading and writing magnetic tapes. As illustrated in Figure 9-1, it includes two reels. The file reel contains the tape to be read or to be written on, and the machine reel contains the tape that has already been processed. The tape is threaded through a read-write head capable of performing the functions of reading and writing; thus, the tape transport unit works very much like a home tape recorder.

As the tape advances past the read-write head, the rated speed of reading or writing is a function of two factors: the recording density of the tape and the linear speed of the tape drive. Thus, if the recording density is 800 bytes per inch (bpi) and the speed is 100 inches per second, the rated read-write speed will be 80,000 bytes per second. Typical ranges are from 40,000 to over 300,000 bytes per second, but much higher speeds also are possible.

A magnetic disk is a magnetically sensitive circular surface resembling a phonograph record. Data are recorded on this disk surface in designated circular

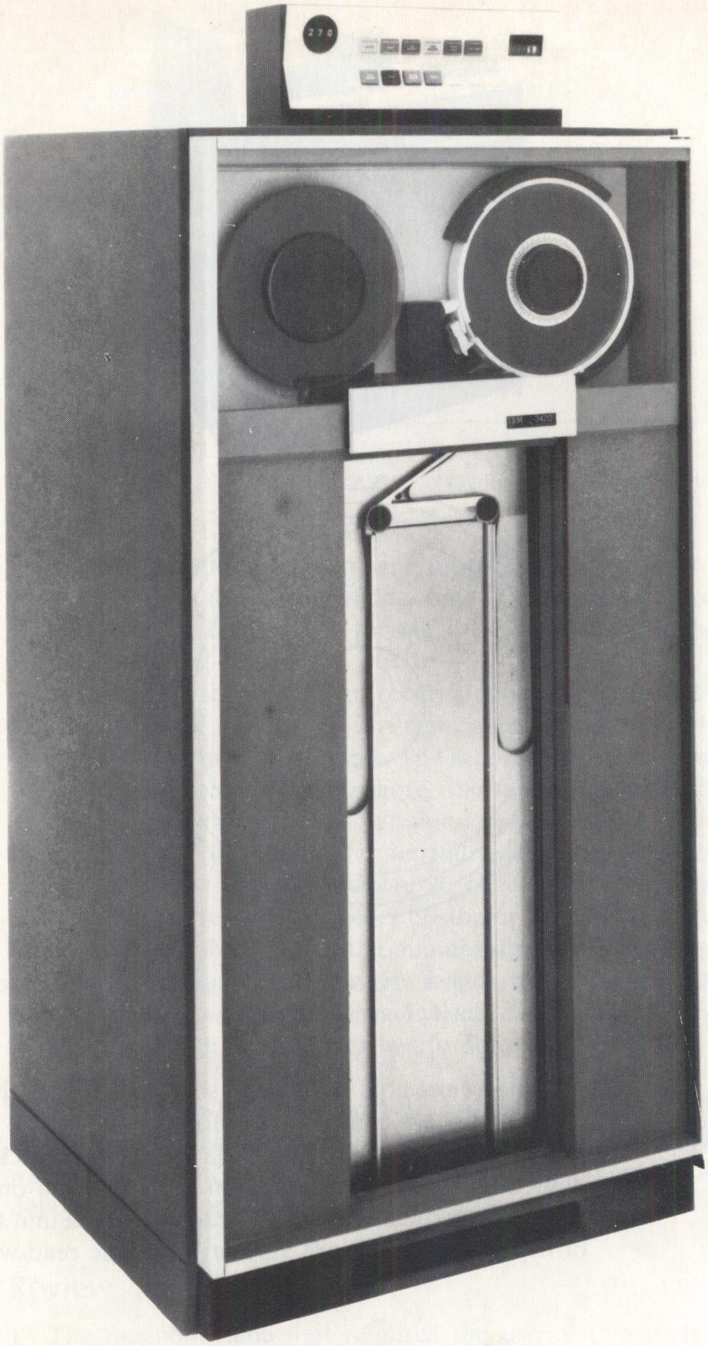


FIGURE 9-1
IBM 3420 MAGNETIC TAPE DRIVE. (IBM CORPORATION.)

bands called tracks. Figure 9-2 portrays a magnetic disk storage device and includes a diagram of tracks on the surface of a disk. Typically, there are several hundred usable tracks on the surface of a magnetic disk. Each track can contain a few thousand characters around its circular length. Each track is separate from the others, and all tracks have the same capacity, even though the circles become smaller as we move away from the periphery. Tracks are referenced in numeric order from 0 up to the last track. Thus, if we have 200 tracks, the first is designated as track 000 and last as track 199.

A disk pack refers to a group of disks stacked on a vertical spindle. The disks are parallel to each other but physically separated from one another. Such packs

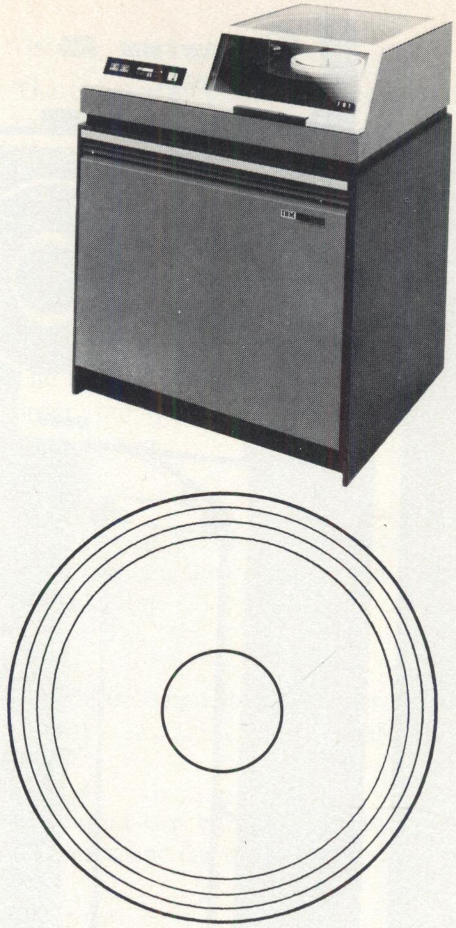


FIGURE 9-2

MAGNETIC DISK STORAGE DEVICE AND SCHEMATIC REPRESENTATION OF TRACKS ON A DISK SURFACE. (IBM CORPORATION.)

may be removable as a unit, so that we can change disk packs for the same reasons that we change reels of magnetic tape.

Typically, for each disk surface in a disk pack there is a read-write head—a device that can read data from or write data on the surface. These heads are fixed to a vertical column and can move as a unit toward or away from the center of the disk pack. Figure 9-3 illustrates the read-write head mechanism.

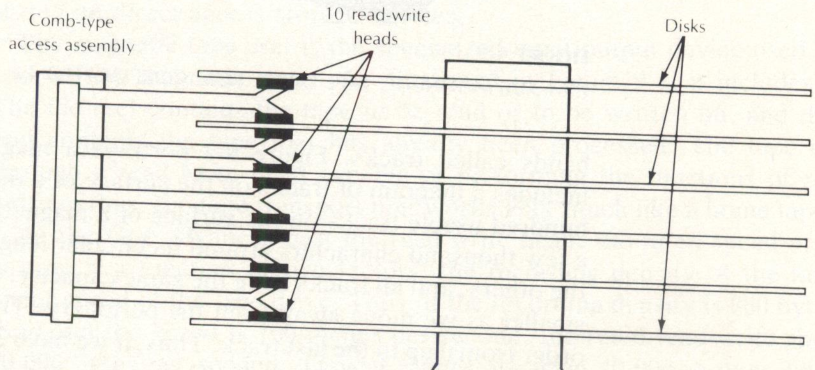


FIGURE 9-3

DISK ACCESS MECHANISM. (IBM CORPORATION.)

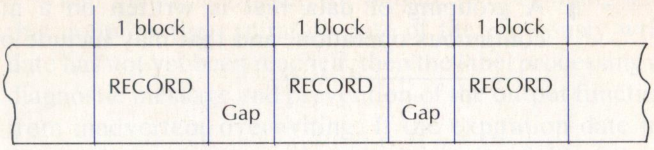


FIGURE 9-4
TAPE RECORDING FORMAT CONTAINING ONE RECORD PER BLOCK.

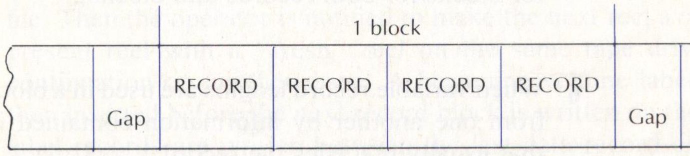


FIGURE 9-5
TAPE RECORDING FORMAT CONTAINING FOUR RECORDS PER BLOCK.

Data are recorded on any magnetic storage device in the form of *blocks*. Each block consists of a grouping of data written (or read) in one continuous operation. Several formats are possible for the data included in the block, as illustrated in Figures 9-4 and 9-5.

In addition to the number of records contained in each block, the size of each record is also of importance. In using magnetic storage, the record size is not restricted to any particular limit. The records can be as long as is suitable for the applications involved. Furthermore, the records can be either fixed or variable in length. An example of a variable-length record is a record containing the transactions for the month for an individual checking account customer. The record for a customer who wrote 20 checks during the month is substantially shorter than the record of another customer who wrote 100 checks. Just as record size can be variable, so can the number of records per block. Overall, then, we can have fixed-length or variable-length records and fixed-length or variable-length blocks. Even though the block size is variable, its size obviously cannot be unlimited. The programmer has to define the maximum limit on block size in each application.

When variable-length records are used, the operating system inserts certain information in a *control field* that usually precedes the record in order to designate where that record ends and the next one begins. The specific format of this field depends on the operating system used.

Review

1 The magnetic tape unit requires the use of two reels when it is used: the machine reel and the file reel. The reel that contains the tape to be written on or read is the ________ reel; the reel that contains the tape that already has been processed is the ________ reel.

file; machine

2 The rated read-write speed of tape units is a function of the recording ________ of the tape and the linear ________ of the tape drive.

density; speed

3 A group of magnetic disks stacked on a vertical shaft in a parallel fashion is called a(n) ________.

disk pack

4 A grouping of data that is written on a magnetic storage device in one continuous operation, and that may include one or more records, is called a ________________.

block

5 In addition to the number of records contained in each block, block size also is determined by the length of each record. Although both the record size and the block size can be fixed, variable size is possible [only for records / only for blocks/for both records and blocks].

for both records and blocks

6 When variable record lengths are used in a block, the records are differentiated from one another by information contained in a ________________ field that usually precedes the record.

control

FILE LABELS

Each magnetic file, such as a reel of tape or a disk pack, is identified by two means. Externally, an adhesive label is attached so that the operator can identify the contents of the file by reading this label. In addition, magnetic labels are included with the file itself. In general, two types of magnetic labels are used: the *header label* and the *trailer label*.

As implied by the name, a *header label* is located at the beginning of the file. This label contains such information as:

File identification: in terms of a file name or a file number
Retention period: the date prior to which the file cannot be overwritten
Creation date: the date the file was established
Sequence number: the sequence number for a multivolume file that consists of several volumes (reels or disk packs).

The header label not only serves as a means of verifying the correct identity of an input file but also is used as a means of preventing inadvertent overwriting on a storage medium used as output. Prior to writing the first block, the header label of an output file is read in, and the retention period is checked against the current date. If the retention period date is later than the current date, a suitable error message will be generated for the operator.

The *trailer label* is a record written as the last record on the file. Typically, it contains the same information as the header label, but in addition it contains a block count, which is a count of the number of data blocks written on the file. This block count is used for control purposes. On a subsequent occasion when this file is input, the number of blocks read in by the program is accumulated and is compared with the block count recorded on the trailer label. If the two counts are not equal, there exists an indication of error.

The trailer label differs depending on whether a file is a single- or a multivolume file. If it is a single-volume file, the trailer is as described above. If it is a multivolume file, the trailer label of each volume indicates the volume (reel or disk pack) number as well.

As far as label processing is concerned, we can differentiate the processing on the basis of whether a file is opened for *output* or *input*.

When a file is used in the output mode, the three main actions are *open, write,* and *close*. The following general operations take place with respect to label processing:

When the file is opened for output, the beginning label is checked for the field that contains the expiration date of the previously written file. If the expiration date has not yet been reached, then the label processing results in some appropriate diagnostic message and prevention of the output function. Thus, a file is protected from inadvertent overwriting. If the expiration date indicates that the previous file can be overwritten, then the beginning field of the new file is written.

Write operations proceed normally until the physical end of the file is reached. If the file is on magnetic tape and it is to be continued on another reel, a trailer label is written indicating that this is the end of the reel and not the end of the file. Then the operator is notified to make the next reel available by replacing the present reel with a "fresh" reel on the same tape drive or by providing for continuation on another drive. A beginning volume label and a header label is then inserted before the next record block is written on the new reel. Thus, three label records are written between the last data record on the first reel and the first data record on the new reel.

If the file is stored on magnetic disk and it is to continue on another disk, the actions parallel those of tape. However, in the case of a disk file, the physical end of the file may be reached without it being the end of the volume. For example, we may have allocated cylinders 21 to 80 to the file, but because of planning error the file requires more than 60 cylinders. In such a case the operating system will issue a diagnostic message to the operator and the job will be abnormally terminated (although some operating systems may allow the machine operator to expand file size immediately).

When a close-file command is carried out, the resulting operation is the insertion of a trailer label at the end of the file. As records are written into the file their number typically is accumulated by the label-processing routine, and this "block count" is written as a field in the trailer label. When the file is used as input on future occasions, the counting is repeated, and the new count is compared to the count in the trailer label to ascertain that all the records in the file have been read by the time the trailer label is processed.

When a file is used in the input mode, the three main actions are *open*, *read*, and *close*. The following general operations take place with respect to label processing:

When the file is opened for input the beginning labels are read to ascertain that the file identification provided in the job control statements agrees with the information in the label record.

The read operation proceeds normally until the trailer label is reached. If the label signifies the end of the file, then the operating system communicates this information to the program (recall READ...AT END...) and aborts the program if a further attempt is made to read from the file. If the file is a multivolume file, the end-of-volume label will signify such a condition and will result in reading and processing the volume and file header labels on the next volume. Of course, the label-processing routine will check to see that the correct sequence of volumes is presented.

When an input file is closed the trailer label is processed to verify that all records have been read. If the file is on tape, the reel is normally rewound as a result of closing the file. In rare instances it is desirable to read a file in reverse order, in which case an "open reverse" command may be issued after a suitable "close with no rewind" command. The trailer label then acts as a header label.

In general, label processing involves a number of specialized functions having to do with proper identification of data files and ensuring the data integrity in such files. As stated earlier, it is a function governed by localized conventions and procedures, and the programmer has to obtain specific instructions from the particular installation.

Review

1 Magnetic files are identified by two general types of labels: ____________ labels for human use and ____________ labels for machine use.

external; magnetic

2 Magnetic labels used with files are of two types. The label included at the beginning of a file is called the ____________ label, while the label at the end of the file is called the ____________ label.

header; trailer

3 For a multivolume file, both the header and trailer labels include a ____________ number.

sequence

4 For a file used in the output mode, the three main actions are ____________, ____________, and ____________.

open; write; close

5 For a file used in the input mode, the three main actions are ____________, ____________, and ____________.

open; read; close

6 The specific procedures associated with label processing [have generally been standardized / are governed by localized conventions].

are governed by localized conventions

COBOL INSTRUCTIONS FOR SEQUENTIAL FILES

There are a number of COBOL instructions which have been designed for use with sequential file processing. In the following sections we present such instructions according to the division of the COBOL program. We then conclude the chapter with two sample programs that illustrate the application of such instructions in some typical tasks.

In keeping with the scope of this book as a first course in COBOL, we do not present and discuss all available options. Instead, we concentrate on those features that are used widely and yet do not require advanced training for their use.

THE FILE-CONTROL SPECIFICATION

For sequential files, the FILE-CONTROL specification in the ENVIRONMENT DIVISION has the format presented in Figure 9-6.

The OPTIONAL specification is used for files that may or may not be present. For example, we could have a file on which we output end-of-month reports only when it is the end of the month. In a program using such a file we would specify the file as OPTIONAL and we include some test in the PROCEDURE DIVISION to determine whether the file is used.

OPTIONAL files must be OPENed in the INPUT, I-O, or EXTEND modes (the latter two modes are discussed later in this chapter).

In the ASSIGN statement, *implementor-name-1* refers to the way each particular operating system designates files. Such designation varies. For example, for IBM mainframes the designation consists of a so-called data-set definition name which

```
FILE-CONTROL.
    SELECT [OPTIONAL] file-name

        ASSIGN TO {implementor-name-1} ...
                  {literal-1         }

    [ [ORGANIZATION IS] SEQUENTIAL]

    [PADDING CHARACTER IS {data-name-1}]
    [                     {literal-2  }]

    [RECORD DELIMITER IS {STANDARD-1         }]
    [                    {implementor-name-2 }]

    [ACCESS MODE IS SEQUENTIAL]

    [FILE STATUS IS data-name-2]
```

FIGURE 9-6
FILE-CONTROL FORMAT FOR SEQUENTIAL FILES.

is associated with a corresponding job control language statement, as in this example:

```
SELECT STUDENT-FILE ASSIGN TO STUFILE
      .
      .
      .
//STUFILE DD DSN = . . . etc. (This is a JCL statement, not a COBOL statement.)
```

In the revised version of COBOL, a file can be ASSIGNED to literal-1, which may be a file name. For example, for a diskette file on the "B:" drive of a personal computer we could have:

```
SELECT STUDENT-FILE ASSIGN TO "B:STUDENT.DAT"
```

The ORGANIZATION clause is optional for sequential files. When it is omitted, as has been done in this book up to now, program execution defaults to ORGANIZATION IS SEQUENTIAL. As one might guess, with indexed and relative files their organization is declared to be INDEXED and RELATIVE, respectively, as will be illustrated in later chapters.

The PADDING CHARACTER choice allows the programmer to specify the character to be used to fill the remainder of a partially filled block. If a block is 6000 bytes long and the records require 200 bytes each, there would be 30 records per block. If the total number of records in the file is 125, the last block would contain 5 data records, and the remaining (30-5) × 200 = 5000 bytes would be "padded" with the specified characters.

Both data-name-1 and literal-2 must be one character long. When a PADDING option is not specified, then the padding character is the default character used by that particular operating system.

The RECORD DELIMITER allows the programmer to either specify the standard or a specific alternative method for delimiting variable length records. Such a method is used to determine the length of each record when records are variable.

The optional ACCESS MODE clause serves mainly a documentation role in the case of sequential files, since SEQUENTIAL is the default case even if the clause is not used. However, for indexed and relative files the clause has a more important role.

The final optional clause, FILE STATUS IS, has to do with exception processing during input or output operations. When executing an I/O operation, there may be exceptions. For example, we may have attempted to close an unopened file, and the like.

As a case in point, consider the AT END clause associated with the READ statement. Its purpose is to provide for specific processing when a particular exception occurs during input, namely, when an end-of-file condition occurs. Other I/O verbs, such as OPEN, WRITE, and CLOSE, do not have clauses paralleling the AT END feature of READ. Yet it is important for the programmer to be able to test for exceptions and treat them under control of program logic. For example, if we attempt to open a file and for some reason the command is not successfully completed, we should be able to recognize the exceptional condition and be able to continue with appropriate processing.

The FILE STATUS clause designates a field into which a code is placed after each I/O statement execution (such as OPEN, CLOSE, READ, WRITE, REWRITE). Let us consider an example:

```
SELECT CUSTOMER-FILE ASSIGN TO file-name
                     ORGANIZATION IS SEQUENTIAL
                     ACCESS MODE IS SEQUENTIAL
                     FILE STATUS IS CUST-FILE-IO-STATUS.
    .
    .
    .
WORKING-STORAGE SECTION.
    .
    .
    .
01  CUST-FILE-IO-STATUS   PIC   X(2).
```

Notice that we chose CUST-FILE-IO-STATUS as the data-name for a two-byte field in WORKING-STORAGE. As each I/O command involving CUSTOMER-FILE is executed, the operating system places a two-byte code in the designated field. Figure 9-7 presents these codes and their meaning.

The FILE STATUS field can then be tested to ascertain the outcome of an I/O operation. For example we may have:

```
OPEN INPUT CUSTOMER-FILE
IF CUST-FILE-IO-STATUS = "00"
    PERFORM READ-CUSTOMER
ELSE
    IF CUST-FILE-IO-STATUS = "10"
        PERFORM NO-MORE-CUSTOMERS
    ELSE
        PERFORM ERROR-ROUTINE.
```

The first IF statement checks to determine if the OPEN command was executed successfully (see Figure 9-7 for the meaning of the codes), while the second IF statement checks for the end-of-file condition.

This approach of using the FILE STATUS feature enables the programmer to have better control over exception processing. The program can test for specific exception conditions and take the appropriate action.

I-O STATUS CODE	EXPLANATION
00	Successful execution.
04	A record whose length is inconsistent with the record description for the file that has been read.
05	A file has been OPENed which is not present.
07	The file storage device is not magnetic tape, yet a CLOSE or OPEN involved a corresponding NO REWIND, REEL, FOR REMOVAL phrase.
10	End-of-file condition.
15	Attempt to read an optional file which is not present.
16	A READ statement was executed while the at end condition is true (attempt to read past the end of the file).
30	A permanent error exists; no further information is available.
34	An attempt is made to write beyond the boundaries of the file.
35	Attempt to OPEN as INPUT, I-O, or EXTEND a nonoptional file.
37	Attempt to OPEN a file that should be on mass storage but is not.
38	Attempt to OPEN a file that has been CLOSED with LOCK.
39	Error during OPEN execution, due to inconsistency between the file description and the actual file.
41	Attempt to OPEN a file that is already opened.
42	Attempt to CLOSE a file that is not open.
43	A successful READ did not precede execution of the current REWRITE command.
44	A boundary violation due to attempt to WRITE or REWRITE a record of improper length.
46	Attempt to READ the next nonexistent record.
47	Attempt to READ from a file not open in INPUT or I-O mode.
48	Attempt to WRITE on a file not open in OUTPUT or EXTEND mode.
49	Attempt to REWRITE on a file not open in I-O mode.
9X	An error condition defined by the particular system in use.

FIGURE 9-7
FILE STATUS CODES AND THEIR MEANING.

Review

1 In the FILE-CONTROL statement, the specification that is included for files that may or may not be present is the ________________ specification.

OPTIONAL

2 The implementor-name-1 in the ASSIGN statement designates a file according to the method of the particular ________________ system being used.

operating

3 When the ORGANIZATION clause is omitted in the FILE-CONTROL statement, program execution defaults to ORGANIZATION is ________.

SEQUENTIAL

4 The clause available in the revised language by which the character that is to be used to fill the remainder of a partially filled block can be specified is the ________________ clause.

PADDING CHARACTER

5 If the optional ACCESS MODE clause is used, the only mode available in the clause is SEQUENTIAL. If the clause is not used, the default mode is the ________________ mode.

SEQUENTIAL

6 The optional clause that makes it possible to test for exceptional conditions during program execution and thereby achieve better program control and the processing of exceptional conditions is the ________ IS clause.

FILE STATUS

THE FILE DESCRIPTION ENTRY

Figure 9-8 presents the general format of the file description entry used for sequential files in the DATA DIVISION

FD marks the beginning of a file description entry and is followed immediately by the name of the file. The name of the file already has been declared in the ENVIRONMENT DIVISION, where it was assigned to a hardware device.

If records are grouped together, the BLOCK CONTAINS clause is used. If each record constitutes one block, the clause may be omitted or the equivalent BLOCK CONTAINS 1 RECORD can be used. When a block contains several records, then the clause must be used. Typically, this clause is used with the RECORDS option. In such a case it references the number of records per block. For example, if we have:

```
FD  PAYROLL-FILE BLOCK CONTAINS 10 RECORDS
```

each block will contain 10 records. However, if the records are of variable size, BLOCK CONTAINS 10 RECORDS will be interpreted to mean the maximum block size. Thus, if the records varied between 10 and 100 characters (this would be so identified in the record description), BLOCK CONTAINS 10 RECORDS would mean that blocks can be as short as $10 \times 10 = 100$ characters in length or as long as $10 \times 100 = 1{,}000$ characters in length. (Actually, the number of

```
FD  file-name

[ BLOCK   CONTAINS   [integer-1 TO] integer-2  { RECORDS    } ]
                                               { CHARACTERS }

[          { CONTAINS integer-3 CHARACTERS                                              } ]
[ RECORD   { IS VARYING IN SIZE [ [FROM integer-4]  [TO integer-5]  CHARACTERS]         } ]
[          {    [DEPENDING on data-name-1]                                              } ]
[          { CONTAINS integer-6 TO integer-7 CHARACTERS                                 } ]

[ LABEL { RECORD IS    }  { STANDARD } ]
[       { RECORDS ARE  }  { OMITTED  } ]

[ VALUE OF { implementor-name-1 IS { data-name-2 } } . . . ]
[                                  { literal-1   }         ]

[ DATA { RECORD IS   }  { data-name-3 } . . . ]
[      { RECORDS ARE }                         ]
```

FIGURE 9-8
FORMAT FOR A FILE DESCRIPTION ENTRY.

characters will be slightly different, depending on the *control fields* used by different compilers in recording variable-length blocks.)

If the option

```
FD  PAYROLL-FILE BLOCK CONTAINS 5 TO 12 RECORDS
```

were used, it would imply that the number of records per block might be a number between 5 and 12, again giving variable block size.

The option BLOCK CONTAINS integer CHARACTERS is used when the block may contain ''padding''—the last part of the block may consist of unusable characters. This could be the case when it is desired that block size be fixed, but, because of variable record length, it would not not be possible to fill the entire block without splitting records across blocks.

The VARYING clause is used to specify variable-length records. Consider this example:

```
FD  CUSTOMER-FILE
            BLOCK CONTAINS 100 to 1000 CHARACTERS
            RECORD IS VARYING IN SIZE
                        FROM 20 to 100 CHARACTERS
                        DEPENDING ON CUST-REC-SIZE
            LABEL RECORDS ARE STANDARD
            DATA RECORD IS CUSTOMER-REC.
01    CUSTOMER-REC.
02    CUSTOMER-NAME PIC X(20).
02    CUSTOMER-ADDRESS PIC X(80).
                        .
                        .
                        .
WORKING-STORAGE SECTION.
                        .
                        .
                        .
01    CUST-REC-SIZE    PIC 999.
```

In the above example, each customer record consists of a 20-byte fixed-length field containing the customer name. A second field in the record, CUSTOMER-ADDRESS, is assumed to be variable in terms of content. A customer whose address is not available would be so designated by placing a 20 in CUST-REC-SIZE, the field identified in the DEPENDING clause. Similarly, for a customer whose address is 65 bytes long, CUST-REC-SIZE would be set to 85. Then, when outputting of data takes place in CUSTOMER-FILE using the WRITE verb, the system strings as many whole records as will fit into each of the 1000 character blocks. When there is insufficient space for one more whole record in the block, then that block is written onto the file, including end-of-block padding.

The reader may wonder how the program can determine the actual address size, given that CUSTOMER-ADDRESS is a fixed length field. One way to proceed is by using the UNSTRING verb to determine the length of the data in CUSTOMER-ADDRESS, minus any trailing spaces. We could redefine the field:

```
02   CUSTOMER-ADDRESS    PIC    X(80).
02 . A   REDEFINES CUSTOMER-ADDRESS.
     03   FILLER   PIC X (79).
     03   LAST-CHAR   PIC   X.
```

Then the following procedure could be used:

```
UNSTRING CUSTOMER-ADDRESS DELIMITED BY "  "
    INTO  XYZ
    COUNT IN CUST-REC-SIZE.
IF CUST-REC-SIZE = 80
    IF LAST-CHAR = SPACE
        SUBTRACT 1 FROM CUST-REC-SIZE.
```

The program segment above reflects the following reasoning. If *two* spaces are used as the delimiter, XYZ would contain the address minus the trailing spaces. If an address were 79 bytes long, however, then the effective delimiter during the unstring operation is the field size, since there would be no two consecutive spaces. We could ascertain if such is the case by testing to determine if the last character (LAST-CHAR) in CUSTOMER-ADDRESS is a space.

Use of variable length records saves file space and has some small impact on the amount of I/O time required to process the file.

Another way of handling variable length records, which is included in the 1974 version of COBOL as well as being in the revised version, uses the third format of the RECORD clause, as in:

```
RECORD CONTAINS 20 TO 100 CHARACTERS
```

In the record description itself we would use the OCCURS . . . DEPENDING ON feature to describe the variable length of the record. (The OCCURS . . . DEPENDING is discussed in Chapter 12, ''Additional Table-Handling Features.'')

The LABEL RECORDS clause is required for all files. The OMITTED option indicates that the file has no beginning or ending label. If the STANDARD option is used, it is understood to be the standard label for the particular computer installation. The natural question may be: Granted that they are ''standard'' for an installation, how do we communicate what the label should be in the context of the program? We said earlier that the label record contains data that identify the file, and obviously each file is identified uniquely. There are two basic ways of saying what the label contents should be. By the first approach (common in IBM computer systems), this information is communicated through program control statements submitted with the COBOL program. In other words, this information is not communicated, strictly speaking, in the COBOL program language. Another way of communicating the contents of label records is by use of the VALUE OF clause. For example, we could have for a particular case:

```
FD  PAYROLL-FILE BLOCK CONTAINS 10 RECORDS
    LABEL RECORDS ARE STANDARD
    VALUE OF IDENTIFICATION IS "A2359"
    RETENTION-PERIOD IS 090
    DATA RECORD IS PAY-REC.
```

In this example, the words IDENTIFICATION and RETENTION-PERIOD are meaningful in a particular installation, and they indicate that the STANDARD label contains a field called IDENTIFICATION, whose content should be A2359. When the file is opened, the field is checked to ascertain that the A2359 data are there; in other words, that the correct file has been mounted. The RETENTION-PERIOD field implies that this file cannot be written on until 90 days have elapsed. Of course, other similar fields are used in the VALUE clause option.

The VALUE clause has been marked as being *obsolete* in the revised version. This means that it should not be used in new programs and that it will be deleted in a future version of COBOL. The ''obsolete'' category allows for a transitional period so that ''old'' features can be phased out of the language. The VALUE clause is not in common use. Most implementors provide means external to COBOL for specifying file label contents.

The DATA RECORD clause identifies the name of the record(s) in the file. For example:

```
            .
            .
            .
DATA RECORDS ARE CHARGE-REC
                 PAY-REC.
```

In the above example, there are two records in the file.

The DATA RECORD clause mainly serves a documentation role, since it is followed immediately by the data record names and the 01 level number. For this reason, the DATA RECORD clause has been marked as being obsolete in the new version and will be deleted in future versions of COBOL.

As a way of summarizing the discussion in this section, the following are examples of file descriptions in the DATA DIVISION:

```
FD  PAYROLL-FILE  BLOCK CONTAINS 10 RECORDS
                  RECORD CONTAINS 80 CHARACTERS
                  LABEL RECORDS ARE OMITTED
                  DATA RECORD IS SAMPLE-REC.

FD  FILE-A        BLOCK CONTAINS 600 CHARACTERS
                  LABEL RECORD IS STANDARD
                      VALUE OF IDENTIFICATION IS "A1-2B"
                  DATA RECORD IS REC.

FD  FILE-B        LABEL RECORDS OMITTED
                  DATA RECORD IS SIMPLE-REC.
```

Review

1 With reference to the sample file descriptions just given, the names of the three files are ________, ________, and ________, respectively.

PAYROLL-FILE; FILE-A; FILE-B

2 The file description just given in which each block contains one record is the [first / second / third] description. The file description in which the record size can be variable is the [first / second / third] description.

third; second

3 The file description that includes a label record is the [first / second / third] description. The file description that includes the optional VALUE OF clause is the [first / second / third] description.

second; second

4 In the revised version of COBOL, the clause in the RECORD statement that is used to specify variable-length records is the ________ clause.

VARYING

THE OPEN AND CLOSE VERBS

The OPEN verb initiates processing of a file and performs appropriate label processing. If the file is to be used as output, when the file is opened the existing

label is checked to ascertain that the previous file can be "scratched." If so, the header label is written on the file. Similarly, if the file is opened for input, its label is read and checked for proper identification.

If an error condition arises as a result of such label processing, it either is reported by the operating system, which normally terminates the program, or it is handled by the program itself through use of the FILE STATUS feature discussed earlier in this chapter.

The OPEN verb has the following format:

```
       { INPUT    file-name [with NO REWIND] }
       { OUTPUT   file-name [with NO REWIND] }
OPEN   { I/O      file-name                  }
       { EXTEND   file-name                  }
```

The INPUT and OUTPUT modes contain the NO REWIND option that can be used with magnetic tape files. Absence of the NO REWIND implies that the reel should be rewound, if necessary, so that it is at its beginning. If the NO REWIND is used, it is assumed that a previous CLOSE . . . WITH NO REWIND has resulted in the tape being positioned at the beginning of the desired file (by being at the end of the previous file that was just closed).

The I/O option can be used only for mass storage files. It allows the program to both input and output on the file. It is used if we want to change a record that we have just read. With tape files, however, we cannot rewrite a record that we have just read in.

The EXTEND option positions the file after its last record. It can be used to add new records at the end of the file and is really a special case of the OUTPUT mode option.

Correct use of input-output verbs depends on the option, or mode, used in the OPEN statement. Table 9-1 summarizes this relationship, with an X designating each permissible combination. For instance, if the INPUT mode option is used then the READ verb can be used, but the WRITE or REWRITE verbs cannot be used.

The CLOSE verb has the following expanded format:

```
                  [ { REEL }  [ WITH NO REWIND ] ]
                  [ { UNIT }  [ FOR REMOVAL    ] ]
CLOSE file-name   [                              ] . . .
                  [        { NO REWIND }         ]
                  [  WITH  { LOCK      }         ]
```

Prior to closing, a file must have been opened. "CLOSE file-name" results in end-of-file procedures. If label records have not been omitted, a trailer label is written and the tape is rewound automatically if it is a tape file. If the option

TABLE 9-1
PERMISSIBLE COMBINATIONS OF OPEN MODE OPTIONS AND INPUT-OUTPUT VERBS

	OPEN MODE			
STATEMENT:	INPUT	OUTPUT	I/O	EXTEND
READ	X		X	
WRITE		X		X
REWRITE			X	

CLOSE file-name REEL is used, this results in closing that reel but not the file as such. Thus, the file itself still will be in an open status. The only circumstance under which the REEL option is used is in the case of multireel files, where the processing of a particular reel for a file may have been completed but other reels for the file still might remain to be processed. If we are using disk files, then the term UNIT is used instead of REEL. The NO REWIND option prevents the rewinding that otherwise is effected automatically by the CLOSE verb. One circumstance in which the user would not want to rewind the tape is when a second file subsequently is to be read from or written on the same tape reel. When the indicated LOCK option is used instead, the file is locked and can be reopened only by restarting the program. The LOCK option thereby serves as protection against accidentally opening and misusing a file whose data have already been processed. The FOR REMOVAL option is used to allow the operator to intervene, remove the reel (at least logically), and replace it with another reel. The specific procedure that should take place in conjunction with using the FOR REMOVAL option is not defined by COBOL; rather, it is determined by the implementor.

Review

1 Processing of a file is initiated by the use of the ________________ verb.

OPEN

2 If the READ verb is to be used subsequent to a file being opened, then the INPUT mode option should be included in the OPEN statement. Similarly, if the WRITE verb is to be used then the ________________ mode option should be included in the OPEN statement.

OUTPUT (or EXTEND)

3 End-of-file procedures are specified by use of the ________________ verb.

CLOSE

4 The option in the closing routine that serves to protect the file from use (and misuse) is the [NO REWIND / LOCK] option, whereas the option that permits further use of the reel is the [NO REWIND / LOCK] option.

LOCK; NO REWIND

THE READ, WRITE, AND REWRITE VERBS

The format for the READ verb is:

```
READ file-name [NEXT] RECORD [INTO identifier]
    [AT END imperative-statement-1]
    [NOT AT END imperative-statement-2]
    [END-READ]
```

This is the same format as we have encountered before, with one small exception. In the revised version of COBOL, NEXT is an option. The NEXT option serves mainly a documentary role for sequential files, but it has a much more important role in indexed and relative files, as will be explained in later chapters.

The AT END clause is optional. In its place we can use the FILE STATUS feature to check whether an end-of-file (as well as any other) condition has occurred while reading.

If a file was designated as OPTIONAL in the SELECT statement and the file is not present, the AT END condition occurs when the first READ is executed. As Table 9-1 indicates, a READ is valid when a file has been opened as INPUT or I/O.

The NOT AT END and the END-READ options apply to the revised language only.

The relevant WRITE format is:

```
WRITE record-name [FROM identifier]
```

As Table 9-1 indicates, the file must be opened as either OUTPUT or EXTEND if the WRITE verb is to be used.

If a BLOCK CONTAINS clause was used in the file description, the system will control automatically the operations needed to form an appropriate block prior to a physical write of the block itself. The programmer need not be concerned about the blocking operation.

The REWRITE statement is a specialized instruction for mass storage files. Its format is:

```
REWRITE record-name [FROM identifier]
```

In order to update a sequential file on disk we may use "OPEN I/O file-name." Then, after issuing a READ command that accesses the record to be updated, we use the REWRITE verb to replace the updated record in the same file instead of WRITE on a different file. With magnetic tape files, we must read a record from one file and MOVE and write the updated record on a new file.

It should be noted that REWRITE can be used only to update an existing file. If we are creating a file, we use the WRITE verb.

Review

1 If the OPEN verb is to be used, the file must be opened in the __________ or __________ mode.

INPUT; I-O

2 If the WRITE verb is to be used, the file must be opened in the __________ or __________ mode.

OUTPUT; EXTEND

3 The verb that is used to update an existing file is the __________ verb. To use this verb, the file must be opened in the __________ mode.

REWRITE; I-O

SAMPLE PROGRAM TO CREATE A SEQUENTIAL FILE

Figure 9-9 illustrates the process of creating a sequential file on magnetic tape or disk. The source records are assumed to be in CUST-SOURCE-FILE. Each record is read in and it is checked for sequence on a field called CUST-NO in columns 1–6. If a record is not in ascending sequence, it is listed on the printer for visual review and correction.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. CREATE-FILE.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT CUST-MAST-FILE ASSIGN TO MASTOUT.
*
     SELECT CUST-SOURCE-FILE  ASSIGN TO READER
            FILE STATUS IS CUST-FILE-STATUS.
*
     SELECT ERROR-FILE  ASSIGN TO PRINTER.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  CUST-MAST-FILE
     LABEL RECORDS ARE STANDARD
     BLOCK CONTAINS 5 RECORDS
     DATA RECORD IS CUST-MAST-REC.
 01  CUST-MAST-REC                     PIC X(75).
*
 FD  CUST-SOURCE-FILE
     LABEL RECORDS OMITTED
     DATA RECORD IS CUST-SOURCE-REC.
 01  CUST-SOURCE-REC.
     02  CUST-NO         PIC 9(5).
     02  CUST-NAME       PIC X(20).
     02  CUST-ADDRESS    PIC X(50).
     02  FILLER          PIC X(5).
*
 FD  ERROR-FILE
     LABEL RECORDS ARE OMITTED
     DATA RECORD IS ERROR-REC.
 01  ERROR-REC                         PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-INDICATOR             PIC XXX VALUE 'NO'.
     88 END-OF-FILE                    VALUE 'YES'.
*
 01  CUST-FILE-STATUS                  PIC XX.
*
 01  PREVIOUS-CUST-NO                  PIC 9(5) VALUE ZERO.
 01  SEQ-ERROR-MESSAGE.
     02  FILLER                 PIC X VALUE SPACE.
     02  FILLER                 PIC X(13) VALUE 'ERROR RECORD:'.
     02  RECORD-OUT             PIC X(75).
*
 01  I-O-ERROR-MESSAGE.
     02 FILLER                  PIC X(25) VALUE
                               'CUST-MASTER I/O EXCEPTION'.
     02 FILLER                  PIC X(16) VALUE
                             '  FILE STATUS = '.
     02 ERR-MESS-FILE-STATUS      PIC XX.
*
 01  JOB-TERMINATOR        PIC XXX VALUE 'NO'.
     88 END-OF-JOB         VALUE 'YES'.
/
 PROCEDURE DIVISION.
*
 100-PROGRAM-SUMMARY.
     OPEN INPUT CUST-SOURCE-FILE
            OUTPUT ERROR-FILE.
```

FIGURE 9-9
SAMPLE PROGRAM TO CREATE A SEQUENTIAL FILE.

```
*
         OPEN OUTPUT CUST-MAST-FILE.
*
         IF CUST-FILE-STATUS = '00'
            PERFORM 200-CREATE-MASTER
         ELSE
            PERFORM 500-I-O-EXCEPTION.
*
         CLOSE CUST-SOURCE-FILE
               ERROR-FILE
               CUST-MAST-FILE.
*
         STOP RUN.
*
     200-CREATE-MASTER.
         PERFORM 300-READ-MASTER.
*
         PERFORM 400-CHECK-WRITE-READ UNTIL END-OF-FILE
                 OR END-OF-JOB.
*
     300-READ-MASTER.
         READ CUST-SOURCE-FILE RECORD.
         IF CUST-FILE-STATUS = '10'
            MOVE 'YES' TO END-OF-FILE-INDICATOR
         ELSE
            IF CUST-FILE-STATUS NOT = '00'
               PERFORM 500-I-O-EXCEPTION
               MOVE 'YES' TO JOB-TERMINATOR.
     400-CHECK-WRITE-READ.
         IF CUST-NO NOT > PREVIOUS-CUST-NO
            MOVE CUST-SOURCE-REC TO RECORD-OUT
            WRITE ERROR-REC FROM SEQ-ERROR-MESSAGE AFTER 2
         ELSE
            WRITE CUST-MAST-REC FROM CUST-SOURCE-REC
            MOVE CUST-NO TO PREVIOUS-CUST-NO.
*
         PERFORM 300-READ-MASTER.
*
     500-I-O-EXCEPTION.
         MOVE CUST-FILE-STATUS TO ERR-MESS-FILE-STATUS
         WRITE ERROR-REC FROM I-O-ERROR-MESSAGE AFTER 1
         MOVE 'YES' TO JOB-TERMINATOR.
*
         READ CUST-SOURCE-FILE RECORD
            AT END MOVE 'YES' TO END-OF-FILE-INDICATOR.
*
         PERFORM 600-READ-CHECK-WRITE UNTIL END-OF-FILE.
*
     600-READ-CHECK-WRITE.
         IF CUST-NO NOT > PREVIOUS-CUST-NO
            MOVE CUST-SOURCE-REC TO RECORD-OUT
            WRITE ERROR-REC FROM SEQ-ERROR-MESSAGE AFTER 1
         ELSE
            WRITE CUST-MAST-REC FROM CUST-SOURCE-REC
            MOVE CUST-NO TO PREVIOUS-CUST-NO.
*
         READ CUST-SOURCE-FILE RECORD
           AT END MOVE 'YES' TO END-OF-FILE-INDICATOR.
```

FIGURE 9-9 *(Continued)*
SAMPLE PROGRAM TO CREATE A SEQUENTIAL FILE.

SEQUENTIAL FILE UPDATING

Files on magnetic tape or disk are maintained or updated to reflect changes that take place. We speak of *master* and *transaction* files as being involved in the updating process.

A *master* file contains reference data that reflect the cumulative status as of a point in time. For example, a payroll master file would contain data on each employee, such as name, address, pay rate, year-to-date earnings, and so forth.

A *transaction* file contains records that either reflect events or indicate changes to the master file. For example, a transaction record at a bank might be a deposit made or a check written. Other examples of transactions would be the addition of a new customer to the master file, deletion of a former customer's name, or a change of the customer's address.

Updating involves processing the transaction file against the master file. The process of updating varies a little, depending on whether we use magnetic tape or disk for storing the master file.

Figure 9-10 illustrates the general procedure involved in updating magnetic tape and disk files. Beginning with the top of the figure, we assume that the transactions originally were recorded manually and then were keyed in and transferred onto a magnetic tape or disk file. The old master file then is processed against the sorted transactions tape to produce the updated master file. In addition, related reports might be produced on the printer.

As indicated by the dotted lines in Figure 9-10, what is now the updated master file becomes the old master file in the next update run, and the old master file from the first run will be used for entry of the updated master file on the next update run. This is known as a parent/child file relationship. However, the procedure we have described results in each case in destruction of the file for the period preceding the one being updated. If we want to have more historical backup, we could use a third file in this procedure, giving rise to a grandparent/parent/child relationship.

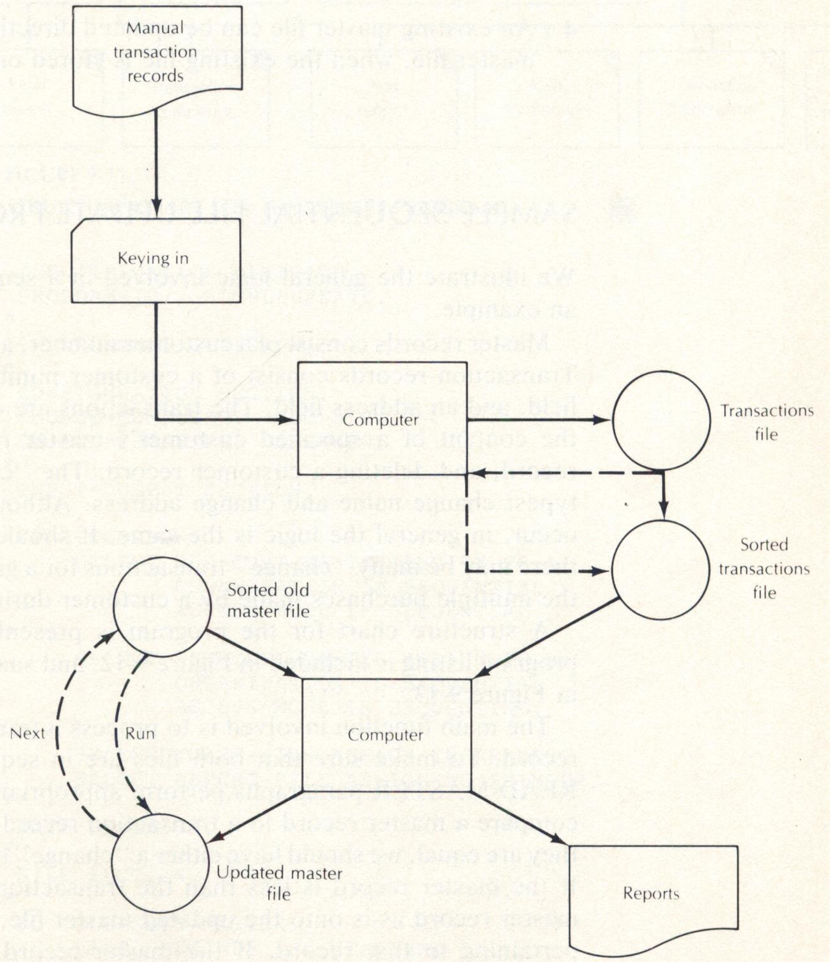


FIGURE 9-10
PROCEDURE USED FOR UPDATING A SEQUENTIAL FILE.

In order to update a master file, both the master file and the transaction file must be sorted on the same basis. Typically, they are sorted according to part number, account number, employee number, or the like.

In the case of a master file stored on disk, we need not create a new file. We simply can REWRITE each updated record in its original location. Thus, it is not necessary to create a new master file each time; however, we should realize that doing so eliminates our backup capability. If something went wrong in the previous update run, we would not find it easy to reconstruct the file and rerun the update program. Thus, even with disk storage it often is preferable to create a new master file.

Review

1 The file that contains cumulative data as of a point in time is called the ______ file.

master

2 The file that contains records that reflect required changes to the master file is the ______ file.

transaction

3 In the process of updating, the updated master file becomes the ______ master file in the next update run.

old

4 An existing master file can be updated directly, rather than by creating a new master file, when the existing file is stored on [tape / disk].

disk

SAMPLE SEQUENTIAL FILE UPDATE PROGRAM

We illustrate the general logic involved in a sequential file update by means of an example.

Master records consist of a customer number, a customer name, and an address. Transaction records consist of a customer number, a transaction code, a name field, and an address field. The transactions are of three general types: changing the content of a specified customer's master record, adding a new customer record, and deleting a customer record. The "change" transactions are of two types: change name and change address. Although many types of changes can occur, in general the logic is the same. It should be noted, however, that often there may be many "change" transactions for a given master record; for example, the multiple purchases made by a customer during a given month.

A structure chart for the program is presented in Figure 9-11, a complete program listing is included in Figure 9-12, and sample input and output are shown in Figure 9-13.

The main function involved is to process a transaction record against a master record. To make sure that both files are in sequence, the READ-TRANS and READ-MASTER paragraphs perform appropriate sequence checking. When we compare a master record to a transaction record, there are three possibilities. If they are equal, we should have either a "change" or a "delete" type of transaction. If the master record is less than the transaction record, then we copy the old master record as-is onto the updated master file, since there are no transactions pertaining to that record. If the master record is higher than the transaction record, then it must be an "add" transaction in order to be valid.

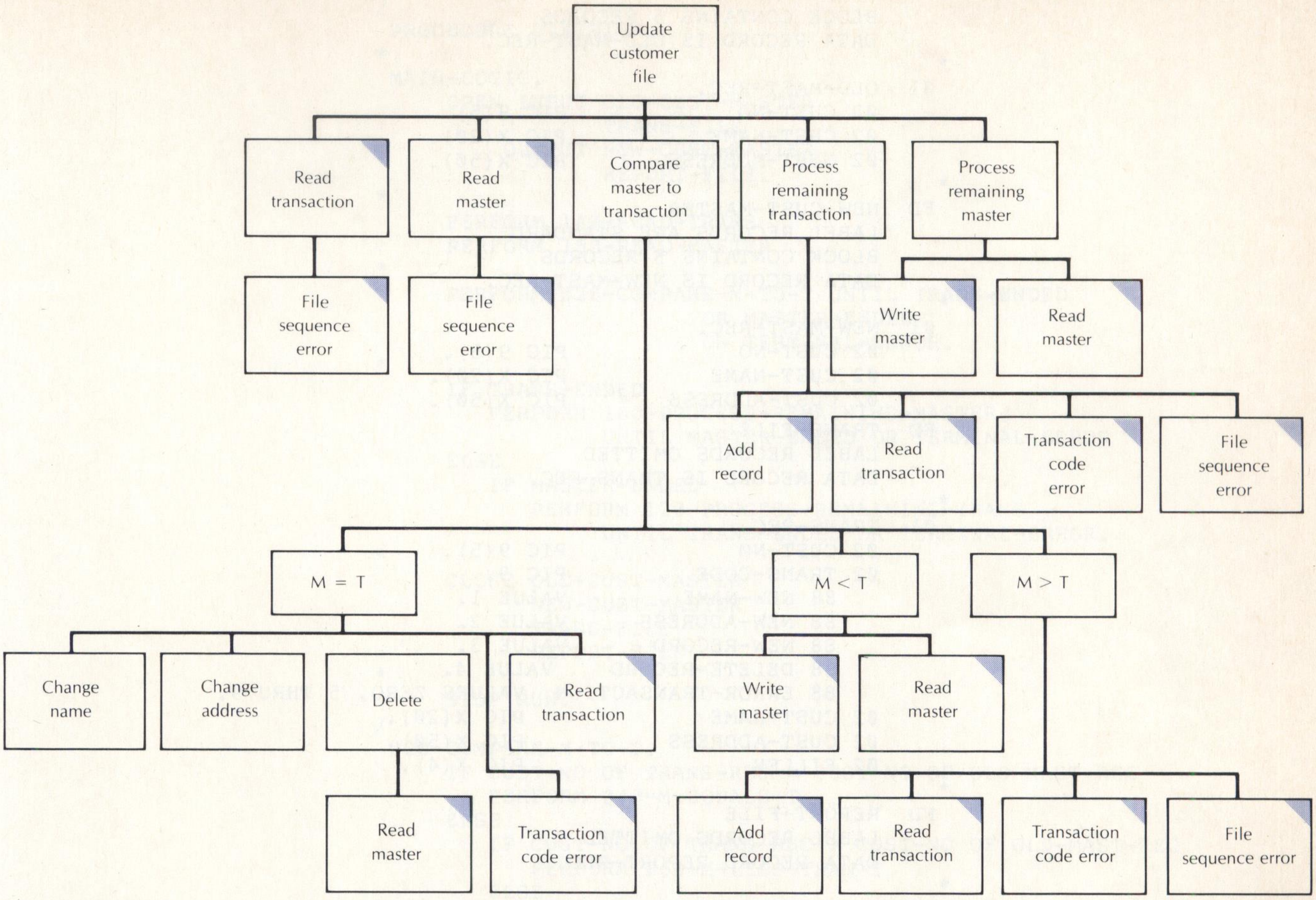


FIGURE 9-11
STRUCTURE CHART FOR THE UPDATE PROGRAM.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  SIMPLEUPDATE.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
     SELECT OLD-CUST-MASTER ASSIGN TO A
            ORGANIZATION IS SEQUENTIAL
            ACCESS MODE IS SEQUENTIAL.
*
     SELECT NEW-CUST-MASTER ASSIGN TO B
            ORGANIZATION IS SEQUENTIAL
            ACCESS MODE IS SEQUENTIAL.
*
     SELECT TRANS-FILE ASSIGN TO READER.
     SELECT REPORT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  OLD-CUST-MASTER
     LABEL RECORDS ARE STANDARD
```

FIGURE 9-12
LISTING OF THE UPDATE PROGRAM.

```
          BLOCK CONTAINS 5 RECORDS
          DATA RECORD IS OLD-MAST-REC.
     *
      Ø1  OLD-MAST-REC.
          Ø2 CUST-NO                PIC 9(5).
          Ø2 CUST-NAME              PIC X(2Ø).
          Ø2 CUST-ADDRESS           PIC X(5Ø).
     *
      FD  NEW-CUST-MASTER
          LABEL RECORDS ARE STANDARD
          BLOCK CONTAINS 5 RECORDS
          DATA RECORD IS NEW-MAST-REC.
     *
      Ø1  NEW-MAST-REC.
          Ø2 CUST-NO                PIC 9(5).
          Ø2 CUST-NAME              PIC X(2Ø).
          Ø2 CUST-ADDRESS           PIC X(5Ø).
      FD  TRANS-FILE
          LABEL RECORDS OMITTED
          DATA RECORD IS TRANS-REC.
     *
      Ø1  TRANS-REC.
          Ø2 CUST-NO                PIC 9(5).
          Ø2 TRANS-CODE             PIC 9.
             88 NEW-NAME            VALUE 1.
             88 NEW-ADDRESS         VALUE 2.
             88 NEW-RECORD          VALUE 3.
             88 DELETE-RECORD        VALUE 4.
             88 ERROR-TRANSACTION    VALUES ZERO, 5 THRU 9.
          Ø2 CUST-NAME                 PIC X(2Ø).
          Ø2 CUST-ADDRESS              PIC X(5Ø).
          Ø2 FILLER                    PIC X(4).
     *
      FD  REPORT-FILE
          LABEL RECORDS OMITTED
          DATA RECORD REPORT-REC.
     *
      Ø1  REPORT-REC                          PIC X(132).
     *
      WORKING-STORAGE SECTION.
     *
      Ø1  END-OF-FILE-SWITCHES.
          Ø2 END-OF-TRANS-SWITCH       PIC XXX VALUE 'NO'.
             88 TRANS-ENDED            VALUE 'YES'.
          Ø2 END-OF-MASTER-SWITCH      PIC XXX VALUE 'NO'.
             88 MASTER-ENDED           VALUE 'YES'.
     *
      Ø1  TERMINATION-SWITCH        PIC XXX VALUE 'NO'.
             88 TERMINAL-ERROR         VALUE 'YES'.
     *
      Ø1  PREVIOUS-REC-VALUES.
          Ø2 PREVIOUS-TRANS-NO         PIC 9(5) VALUE ZEROS.
          Ø2 PREVIOUS-MAST-NO          PIC 9(5) VALUE ZEROS.
     *
      Ø1  ERROR-MESSAGE-RECORD.
          Ø2 FILLER                    PIC X VALUE SPACE.
          Ø2 MESSAGE-FIELD             PIC X(75).
     *
      Ø1  PREVIOUS-VALUES-OUT.
          Ø2 FILLER                    PIC X VALUE SPACE.
          Ø2 FILLER                    PIC X(18)
                                       VALUE 'PREVIOUS TRANS NO '.
          Ø2 TRANS-NO-OUT              PIC 9(5).
          Ø2 FILLER                    PIC X(3) VALUE SPACES.
          Ø2 FILLER                    PIC X(17) VALUE
                                       'PREVIOUS MAST NO '.
          Ø2 MAST-NO-OUT               PIC 9(5).
```

FIGURE 9-12 *(Continued)*
LISTING OF THE UPDATE PROGRAM.

```
/
 PROCEDURE DIVISION.
*
 MAIN-LOGIC.
     OPEN INPUT OLD-CUST-MASTER
                TRANS-FILE
          OUTPUT NEW-CUST-MASTER
                REPORT-FILE.
*
     PERFORM 140-READ-TRANS
     PERFORM 160-READ-MASTER
*
     PERFORM 020-COMPARE-M-TO-T UNTIL TRANS-ENDED
                          OR MASTER-ENDED
                          OR TERMINAL-ERROR.
*
     IF TRANS-ENDED
         PERFORM 100-PROCESS-REMAINING-MASTER
                UNTIL MASTER-ENDED OR TERMINAL-ERROR
     ELSE
         IF MASTER-ENDED
             PERFORM 120-PROCESS-REMAINING-TRANS
                UNTIL TRANS-ENDED OR TERMINAL-ERROR.
*
     CLOSE OLD-CUST-MASTER
           NEW-CUST-MASTER
           TRANS-FILE
           REPORT-FILE
*
     STOP RUN.
/
 020-COMPARE-M-TO-T.
     IF CUST-NO OF TRANS-REC = CUST-NO OF OLD-MAST-REC
         PERFORM 040-M-EQUALS-T
     ELSE
         IF CUST-NO OF TRANS-REC > CUST-NO OF OLD-MAST-REC
             PERFORM 060-M-LESS-THAN-T
         ELSE
             PERFORM 080-M-GREATER-THAN-T.
*
 040-M-EQUALS-T.
     IF NEW-NAME
         PERFORM 180-NAME-CHANGE
     ELSE
         IF NEW-ADDRESS
             PERFORM 200-ADDRESS-CHANGE
         ELSE
            IF DELETE-RECORD
                PERFORM 220-DELETION
            ELSE
                PERFORM 300-TRANS-CODE-ERROR.
*
     PERFORM 140-READ-TRANS.
 060-M-LESS-THAN-T.
     PERFORM 260-WRITE-MASTER
     PERFORM 160-READ-MASTER.
*
 080-M-GREATER-THAN-T.
     IF NEW-RECORD
         PERFORM 240-ADD-RECORD
         PERFORM 140-READ-TRANS
     ELSE
         IF ERROR-TRANSACTION
             PERFORM 300-TRANS-CODE-ERROR
             PERFORM 140-READ-TRANS
         ELSE
             MOVE 'FILES OUT OF SEQUENCE' TO MESSAGE-FIELD
             PERFORM 280-FILE-SEQUENCE-ERROR.
```

FIGURE 9-12 *(Continued)*
LISTING OF THE UPDATE PROGRAM.

```
*
 100-PROCESS-REMAINING-MASTER.
     PERFORM 260-WRITE-MASTER
     PERFORM 160-READ-MASTER.
*
 120-PROCESS-REMAINING-TRANS.
     IF NEW-RECORD
        AND CUST-NO OF TRANS-REC > PREVIOUS-MAST-NO
            PERFORM 240-ADD-RECORD
            PERFORM 140-READ-TRANS
     ELSE
        IF ERROR-TRANSACTION
            PERFORM 300-TRANS-CODE-ERROR
            PERFORM 140-READ-TRANS
        ELSE
            MOVE 'FILES OUT OF SEQUENCE' TO MESSAGE-FIELD
            PERFORM 280-FILE-SEQUENCE-ERROR.
*
 140-READ-TRANS.
     READ TRANS-FILE RECORD
          AT END MOVE 'YES' TO END-OF-TRANS-SWITCH.
*
     IF TRANS-ENDED
          NEXT SENTENCE
     ELSE
        IF CUST-NO OF TRANS-REC < PREVIOUS-TRANS-NO
           MOVE 'TRANSACTION FILE OUT OF SEQUENCE'
                                   TO MESSAGE-FIELD
           PERFORM 280-FILE-SEQUENCE-ERROR
        ELSE
           MOVE CUST-NO OF TRANS-REC TO PREVIOUS-TRANS-NO.
*
 160-READ-MASTER.
     READ OLD-CUST-MASTER RECORD INTO NEW-MAST-REC
         AT END MOVE 'YES' TO END-OF-MASTER-SWITCH.
     IF MASTER-ENDED  NEXT SENTENCE
     ELSE
        IF CUST-NO OF OLD-MAST-REC NOT > PREVIOUS-MAST-NO
           MOVE 'MASTER FILE OUT OF SEQUENCE' TO MESSAGE-FIELD
           PERFORM 280-FILE-SEQUENCE-ERROR
        ELSE
           MOVE CUST-NO OF OLD-MAST-REC TO PREVIOUS-MAST-NO.
*
 180-NAME-CHANGE.
     MOVE CUST-NAME OF TRANS-REC TO CUST-NAME OF NEW-MAST-REC.
*
 200-ADDRESS-CHANGE.
     MOVE CUST-ADDRESS OF TRANS-REC
         TO CUST-ADDRESS OF NEW-MAST-REC.
*
 220-DELETION.
     PERFORM 160-READ-MASTER.
*
 240-ADD-RECORD.
     IF CUST-NO OF TRANS-REC > CUST-NO OF NEW-MAST-REC
        PERFORM 260-WRITE-MASTER
        PERFORM 160-READ-MASTER.
*
     MOVE CUST-NO OF TRANS-REC TO CUST-NO OF NEW-MAST-REC
     MOVE CUST-NAME OF TRANS-REC TO CUST-NAME OF NEW-MAST-REC
     MOVE CUST-ADDRESS OF TRANS-REC
          TO CUST-ADDRESS OF NEW-MAST-REC.
*
     PERFORM 260-WRITE-MASTER
*
     MOVE OLD-MAST-REC TO NEW-MAST-REC.
*
* THE ABOVE MOVE RESTORES THE CONTENTS OF NEW-MAST-REC
* WHICH WERE DESTROYED BY THE NEW RECORD JUST ADDED.
* RECALL THEAT IN 160-READ-MASTER PARAGRAPH AS EACH OLD-MAST-REC
* IS READ IN, IT IS MOVED INTO NEW-MAST-REC.
```

FIGURE 9-12 *(Continued)*

LISTING OF THE UPDATE PROGRAM.

```
      *
       260-WRITE-MASTER.
           WRITE REPORT-REC FROM NEW-MAST-REC
                    AFTER ADVANCING 1 LINE
      *
           WRITE NEW-MAST-REC.
      /
       280-FILE-SEQUENCE-ERROR.
           WRITE REPORT-REC FROM ERROR-MESSAGE-RECORD
                    AFTER ADVANCING PAGE.
      *
           MOVE 'TRANSACTION RECORD AT TIME OF ERROR'
                    TO MESSAGE-FIELD
           WRITE REPORT-REC FROM ERROR-MESSAGE-RECORD
                    AFTER ADVANCING 2 LINES
      *
           MOVE TRANS-REC TO MESSAGE-FIELD
           WRITE REPORT-REC FROM ERROR-MESSAGE-RECORD
                    AFTER ADVANCING 2 LINES
      *
           MOVE 'MASTER RECORD AT TIME OF ERROR'
                    TO MESSAGE-FIELD
           WRITE REPORT-REC FROM ERROR-MESSAGE-RECORD
                    AFTER ADVANCING 2 LINES
           MOVE OLD-MAST-REC TO MESSAGE-FIELD
           WRITE REPORT-REC FROM ERROR-MESSAGE-RECORD
                    AFTER ADVANCING 2 LINES
      *
           MOVE PREVIOUS-TRANS-NO TO TRANS-NO-OUT
           MOVE PREVIOUS-MAST-NO TO MAST-NO-OUT
           WRITE REPORT-REC FROM PREVIOUS-VALUES-OUT
      *
           MOVE 'YES' TO TERMINATION-SWITCH.
      *
       300-TRANS-CODE-ERROR.
           MOVE SPACES TO REPORT-REC
           WRITE REPORT-REC
                   AFTER ADVANCING 1 LINE
           MOVE 'THE FOLLOWING TRANSACTION RECORD CONTAINS A CODE ERROR'
                   TO MESSAGE-FIELD
           WRITE REPORT-REC
                   AFTER ADVANCING 1 LINE
           WRITE REPORT-REC FROM TRANS-REC
                   AFTER ADVANCING 1 LINE
           MOVE SPACES TO REPORT-REC
           WRITE REPORT-REC
                   AFTER ADVANCING 1 LINE.
```

FIGURE 9-12 *(Continued)*
LISTING OF THE UPDATE PROGRAM.

At some point in the processing, one of the two files runs out of data (or both could run out at the same time). If no more transactions remain, we simply copy the rest of the old master onto the new master. If no more master records remain, then the remaining transactions should be "add" transactions and therefore should be higher than the last master record processed.

Two kinds of error messages are produced in this sample program. The first one is generated if the files are found to be out of sequence. We then terminate the program, printing first the records involved and then the immediately preceding records, for reference. It is not always necessary to terminate the program, but in this case it seems the reasonable thing to do. The second type of error has to do with incorrect transaction codes. When such errors occur, we write an error message on a separate file, which we assume also is ASSIGNed to the printer device.

The printed output of this program is a simple listing of the updated file. Generally, there would be a list of updated, new, and deleted records printed, rather than the complete file. Exercise 9.2 at the end of this chapter asks for such a modification to the program.

```
Original Master-File Data

11111NAME-1                    ADDRESS-1
22222NAME-2                    ADDRESS-2
33333NAME-3                    ADDRESS-3
44444NAME-4                    ADDRESS-4
10000SEQUENCE ERROR
55555NAME-5                    ADDRESS-5
60000NAME-6

Transaction Data

222221NEW-NAME
234563NEW-CUSTOMER              ADDRESS OF NEW
4000032ND-NEW-CUSTOMER          ADDRESS OF 2ND NEW
444444 DELETE EXAMPLE
555555WRONG CODE EXAMPLE
555554                          NEW ADDRESS
543211 SEQ ERROR
666663 3RD NEW

Output from Update Program

11111NAME-1                    ADDRESS-1
22222NEW-NAME                  ADDRESS-2
23456NEW-CUSTOMER              ADDRESS OF NEW
33333NAME-3                    ADDRESS-3
400002ND-NEW-CUSTOMER          ADDRESS OF 2ND NEW

555555WRONG CODE EXAMPLE

 TRANSACTION FILE OUT OF SEQUENCE
 TRANSACTION RECORD AT TIME OF ERROR
 543211 SEQ ERROR
 MASTER RECORD AT TIME OF ERROR
 60000NAME-6
 PREVIOUS TRANS NO 55555    PREVIOUS MAST NO 60000
```

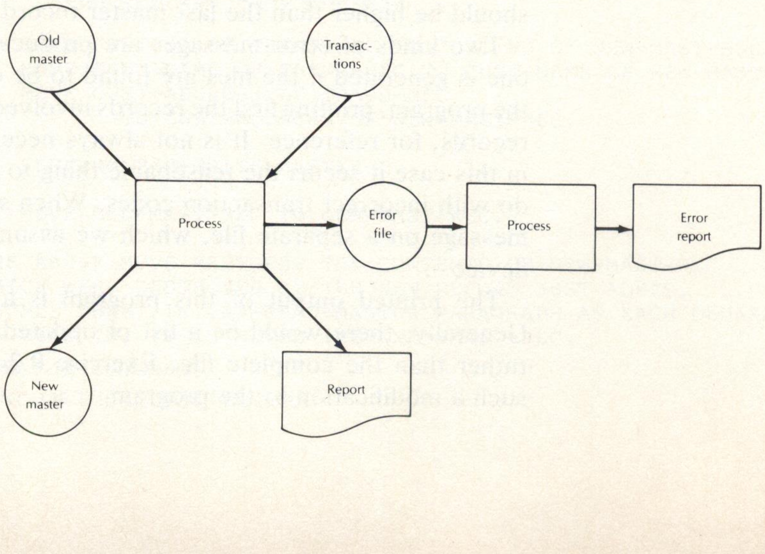
FIGURE 9-13

SAMPLE INPUT AND OUTPUT FOR THE UPDATE PROGRAM.

The program is self-documenting and the reader should be able to follow it. This program can be used as the basis for most sequential updating programs because the basic logic of all such programs is very similar.

EXERCISES

9.1 Modify the sample update program in Figure 9-12 so that transaction error records are saved on a tape or disk file and are printed after the update process has been completed. In a process chart form, we should have:



9.2 Modify the sample update program in Figure 9-12 so that during the update process only those master records that were in some way altered are output on the printer. Where appropriate, show both the old and the new record for visual reference and comparison. If a record is added or deleted, however, only one master record is relevant. The report should have the following format:

Changes to Customer Master MM/DD/YR	
RECORD	ACTION
OLD: xxxxxxxxxxxxxx	CHANGED ADDRESS
NEW: xxxxxxxxxxxxxx	
NEW: xxxxxxxxxxxxxx	NEW RECORD ADDED
OLD: xxxxxxxxxxxxxx	DELETED RECORD
OLD: xxxxxxxxxxxxxx	CHANGED NAME
NEW: xxxxxxxxxxxxxx	

9.3 Write a program to update a simplified payroll master file. The master is sequenced on employee number, and records have the following layout:

Employee number	PIC 9(5)
Employee name	PIC X(20)
Pay rate	PIC 9(2)V99
Withholding rate	PIC V99
Year-to-date gross pay	PIC 9(5)V99
Year-to-date net pay	PIC 9(5)V99

The transaction records are of two types. The first type contains changes to the master file and has the following layout:

Employee number	PIC 9(5)
Employee name	PIC X(20)
Transaction code	PIC 9
1 = Create new record	
2 = Delete record	
3 = Change pay rate	
Pay rate	PIC 9(2)V99
Withholding rate	PIC V99

The second type of transaction record is *daily* time cards that have the following layout:

Employee number	PIC 9(5)
Employee name	PIC X(20)
Transaction code	PIC 9
4 = Time card	
Hours worked	PIC 9(2)V99

In addition to the updated master file, the program should produce a printed report as follows:

		THIS WEEK		YEAR-TO-DATE	
EMPLOYEE NO.	EMPLOYEE NAME	GROSS	NET	GROSS	NET
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
TOTAL PAYROLL		XXXX	XXXX	XXXX	XXXX

For each employee, this week's gross pay is computed by multiplying pay-rate times the hours worked for the whole week (remember that time cards are daily records). If the hours worked exceed 40, time-and-a-half is paid for the hours over 40; however, no employee should work more than 10 hours in a given day nor more than 48 hours in a given week. Either condition would be an error, and the employee is not paid. Instead, an error message is printed.

Net pay is computed by multiplying gross pay by the withholding rate and subtracting this amount from the gross pay.

The year-to-date gross and net on the report include the addition of this week's gross and net pay to the previously accumulated amounts.

9.4 A company maintains inventory data on a master file sorted on part number. Each record contains the following types of data for each item held in inventory:

FIELD	SIZE
Part number	5 numeric positions
Part name	15 alphanumeric positions
Quantity	5 numeric positions

For the sake of simplicity, there are two types of transactions: receipts and issues. Each transaction has the following format:

FIELD	SIZE
Part number	5 numeric positions
Transaction code 1 = receipt 2 = issue	1 numeric position
Quantity	5 numeric positions

Batches of transaction records are accumulated and then processed to update the master file and to print a report that lists each part number, name, previous quantity balance, and new balance. When the transaction code is 1, the quantity is added; if the code is 2, the quantity is subtracted.

a Write a program to create the master file. Sample input for the master tape is as follows:

```
035611/2 HP EL MOTOR02000
10513TRANSFORMER    08000
30561GEAR TRAIN-A   07890
30562GEAR TRAIN-B   10250
30564GEAR TRAIN-C   04650
30579GEAR TRAIN-G   08529
40100STEEL PLATE-1A 06099
40110STEEL PLATE-2A 00852
40120STEEL PLATE-3A 00996
40130STEEL PLATE-4A 01250
40140STEEL PLATE-5B 02899
40150STEEL PLATE-3C 08192
51000BRASS FTNGS-A  12695
51020BRASS FTNGS-B  08569
51030BRASS FTNGS-C  09992
60256BALL BEARING-A201695
60257BALL BEARING-A302561
```

60258BALL BEARING-A410883
60259BALL BEARING-A513429
60260BALL BEARING-A608866
60261BALL BEARING-A706219

b Write a program to update the master file, given a set of transaction records. The program should perform a sequence check to ascertain that the transaction input records are in the same sequence as the master records. It is possible that some items may have no corresponding transactions, but no transactions are present for items not on the master file. Sample input for transactions is as follows (the master file input is the same as shown in part a):

10513200200
10513110000
30562200500
30562200800
30562200900
30564108000
40100112000
40100204000
40100203000
40140110000
51030200200
51030200965
60261200600
60261200500
60261200900
60261104000

Sample output is presented in Figure 9-14.

9.5 The county assessor's office maintains a master file of property owners, in the following (simplified) format:

Lot number	9-digit code
Owner name	26 alphanumeric characters
Assessed valuation	8-digit field, including 2 decimal places

ITEM NUMBER	PART NAME	PREVIOUS BALANCE	NEW BALANCE
03561	1/2 HP EL MOTOR	2000	2000
10513	TRANSFORMER	8000	17800
30561	GEAR TRAIN-A	7890	7890
30562	GEAR TRAIN-B	10250	8050
30564	GEAR TRAIN-C	4650	12650
30579	GEAR TRAIN-G	8529	8529
40100	STEEL PLATE-1A	6099	11099
40110	STEEL PLATE-2A	0852	0852
40120	STEEL PLATE-3A	0996	0996
40130	STEEL PLATE-4A	1250	1250
40140	STEEL PLATE-5B	2899	12899
40150	STEEL PLATE-3C	8192	8192
51000	BRASS FTNGS-A	12695	12695
51020	BRASS FTNGS-B	8569	8569
51030	BRASS FTNGS-C	9992	8827
60256	BALL BEARING-A2	1695	1695
60257	BALL BEARING-A3	2561	2561
60258	BALL BEARING-A4	10883	10883
60259	BALL BEARING-A5	13429	13429
60260	BALL BEARING-A6	8866	8866
60261	BALL BEARING-A7	6219	8219

FIGURE 9-14

SAMPLE OUTPUT FOR EXERCISE 9-4.

An update run involves reading transaction records and creating an updated master file. The input records have the following format:

Lot number	9-digit code
Code	1-digit code
1 = change owner name	
2 = change assessed valuation	
3 = change both owner and valuation	
4 = add to tax rolls	
9 = remove from tax rolls	
Owner name	26 characters
Assessed valuation	8 digits in dollars and cents

If a code of 1 is used, the valuation field is blank. If a code of 2 is used, the owner field is blank. A code of 9 implies that the record is blank from column 11 on.

The printer report should have the following approximate layout:

LOT NUMBER	OWNER	ASSESSED VALUE	NEW OWNER	NEW ASSESSMENT	OFF ROLLS
XXXXXXXXX	XXXXXXXX	$ XXXXXX.XX	XXXXXXXXX	$ XXXXXX.XX	
XXXXXXXXX	XXXXXXXX	XXXX.XX			***

Whenever an item is eliminated, it is signaled by three asterisks in the OFF ROLLS column.

Write a program to update such a file. The program should check for correct sequence in the transaction and master records and for possibly erroneous codes in the transactions.

The sample master file is as follows:

LOT NUMBER	OWNER NAME	ASSESSED VALUATION
000150000	JENKING, ANTHONY	10,872.00
000180000	ANDREWS, JULIA	256,237.00
000290000	THOMAS, THEODORE	162,116.00
000350000	MCDONALD, DONNA	769,276.00
000720000	MARTIN, JANE	99,998.00
001050000	RICHARDSON, PETER	820,600.00
001120000	SILVA, ROBIN	959,999.00

Sample transaction records are:

LOT NUMBER	CODE	OWNER NAME	ASSESSED VALUATION
000180000	1	ANDREWS, THOMAS	
000290000	2		300,000.00
000720000	3	STEINMAN, WILLA	100,000.00
001050000	9		

Sample output is as follows:

LOT NUMBER	OWNER	ASSESSED VALUE	NEW OWNER	NEW ASSESSMENT	OFF ROLLS
00015000	JENKING, ANTHONY	$ 10,872.00			
000180000	ANDREWS, JULIA	$256,237.00	ANDREWS, THOMAS		
000290000	THOMAS, THEODORE	$162,116.00		$300,000.00	
000350000	MCDONALD, DONNA	$769,276.00			
000720000	MARTIN, JANE	$ 99,998.00	STEINMAN, WILLA	$100,000.00	
001050000	RICHARDSON, PETER	$820,600.00			***
001120000	SILVA, ROBIN	$959,999.00			

10 SORTING AND MERGING SEQUENTIAL FILES

- COBOL SORT FEATURE
- SORT STATEMENT FORMATS
- FILE MERGING
- EXERCISES

COBOL SORT FEATURE

In our description of sequential file maintenance in the preceding chapter, it was evident that sequential files must be sorted in sequence. The basis for the sequence is dependent on the situation. For example, in the processing of sales transactions, analysis may be required by product, by date, and by geographic area. In order to produce such a report, the data file has to be sorted so that the data are sequenced in the desired order.

Since file sorting frequently is a common need, the COBOL language incorporates a sort feature that makes it possible to accomplish this operation with minimal programming. The programmer need not be concerned with the details of any sort algorithm in using this feature, but may simply specify the files to be sorted, the sort key (or keys) to be used, and any special procedures for the handling of files before or after the sort. We illustrate here the COBOL sort feature by means of two examples.

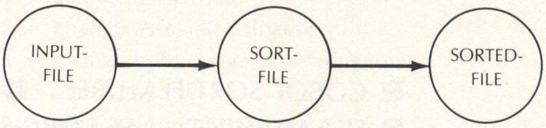
Example 1

Assume that we have a sequential file with the following record description in the DATA DIVISION:

```
01  INPUT-RECORD.
    02  ACCOUNT-NUMBER      PICTURE 9(8).
    02  NAME                PICTURE X(20).
    02  TRANSACTION-DATE.
        03  DAY-OF-YEAR     PICTURE 999.
        03  YEAR            PICTURE 99.
    02  OTHER-DATA          PICTURE X(47).
```

Suppose we wish to sort the file in ascending sequence according to ACCOUNT-NUMBER and in descending sequence according to YEAR. That is, for each

account, all records are to be arranged from the most recent to the least recent YEAR. Also assume that the sorted file is to be called SORTED-FILE. The sorting process can be portrayed as involving three files, INPUT-FILE, SORT-FILE, and SORTED-FILE, as follows:



Because the sorting procedure is preprogrammed, you need not be concerned about the detail of the SORT-FILE.

In concept, SORT-FILE represents a preprogrammed file whose description has been embedded in the sorting routine. As the above diagram illustrates, data from the INPUT-FILE are transferred to the SORT-FILE where they are sorted, and the sorted data are then output onto the SORTED-FILE (these specific file-names are, of course, arbitrary choices).

The SORT-FILE is a conceptual file which may involve several physical files. Typically, the sort routine uses several sequential files in order to execute the sort. Since the sort routine is automated, however, the COBOL programmer describes the file as if it were one physical file, and it is through JCL statements that we describe the physical structure of the file.

Figure 10-1 presents the COBOL program that can be used to sort the file described in this example problem. In the ENVIRONMENT DIVISION, three files are identified in the SELECT statements. Notice that SORT-FILE has been ASSIGNed as if it were one physical file.

In the DATA DIVISION, the INPUT-FILE is described in the usual fashion. However, the SORT-FILE is introduced with the special SD level indicator (which stands for Sort Descriptor). The SD level indicator specifies that this is a file to be used in conjunction with the sort routine. Notice that there is no LABEL clause given for such a file.

As far as the record description for the SORT-FILE is concerned, it is just like any other such description. We used the same description as we used for the INPUT-FILE, and qualification is used to differentiate between the two records.

Finally, the SORTED-FILE record has been described as one field of 80 characters to illustrate one possible variation. Since the program is concerned only with the sorting of the file, there is no need to describe the records that constitute this file.

The relevant PROCEDURE DIVISION is simple and consists of just one paragraph. The SORT verb is very powerful in that the programmer need only specify the sort keys and the source and destination of the file records. The statement SORT SORT-FILE identifies the name of the file to be sorted—which should be the same file introduced by an SD entry in the DATA DIVISION. The ASCENDING KEY ACCOUNT-NUMBER OF SORT-RECORD clause specifies that the file is to be sorted in ascending ACCOUNT-NUMBER OF SORT-RECORD sequence. The DESCENDING KEY YEAR OF SORT-RECORD clause specifies that, within each ACCOUNT-NUMBER OF SORT-RECORD, we wish to sort in descending sequence with respect to the values contained in the YEAR OF SORT-RECORD field. The key written first is the principal basis for the sort. Other keys are of decreasing sorting significance as we proceed from one to the next. For example, consider the following KEY clauses:

```
ASCENDING KEY STATE-NAME
ASCENDING KEY COUNTY-NAME
ASCENDING KEY CITY-NAME
```

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SORT1.
*
REMARKS.  THIS PROGRAM ILLUSTRATES SORTING A SEQUENTIAL
    FILE CALLED INPUT-FILE AND MAKING THE SORTED FILE
    AVAILABLE IN SORTED-FILE.
*
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-FILE  ASSIGN TO READER.
    SELECT SORT-FILE   ASSIGN TO SORTWORK.
    SELECT SORTED-FILE ASSIGN TO PRINTER.
*
DATA DIVISION.
*
FILE SECTION.
FD  INPUT-FILE LABEL RECORD OMITTED
             DATA RECORD IS INPUT-RECORD.
*
01  INPUT-RECORD.
    02  ACCOUNT-NUMBER     PIC 9(8).
    02  NAME               PIC X(20).
    02  TRANSACTION-DATE.
        03  DAY-OF-YEAR    PIC 999.
        03  YEAR           PIC 99.
    02  OTHER-DATA         PIC X(47).
*
SD  SORT-FILE DATA RECORD IS SORT-RECORD.
*
01  SORT-RECORD.
    02  ACCOUNT-NUMBER     PIC 9(8).
    02  NAME               PIC X(20).
    02  TRANSACTION-DATE.
        03  DAY-OF-YEAR    PIC 999.
        03  YEAR           PIC 99.
    02  OTHER-DATA         PIC X(47).
*
FD  SORTED-FILE LABEL RECORD OMITTED
       DATA RECORD IS SORTED-RECORD.
*
01  SORTED-RECORD          PIC X(80).
PROCEDURE DIVISION.
*
SORTING-PARAGRAPH.
    SORT SORT-FILE
*
          ON ASCENDING KEY ACCOUNT-NUMBER OF SORT-RECORD
             DESCENDING KEY YEAR OF SORT-RECORD
*
          USING INPUT-FILE
*
          GIVING SORTED-FILE.
*
    STOP RUN.
```

FIGURE 10-1

SAMPLE SORT PROGRAM FOR EXAMPLE 1.

The order of listing of these clauses indicates that STATE-NAME is the principal basis for the sort. Put another way, CITY-NAME will be sorted within COUNTY-NAME, and COUNTY-NAME will be sorted within STATE-NAME. The fields used for the sort order are often referred to as "sort keys," and the terms "major," "intermediate," and "minor" sort keys are used to describe multiple sort keys. Thus in the above example STATE-NAME is the major sort

key, COUNTY-NAME is the intermediate sort key, and CITY-NAME is the minor key. Of course if we have more than three sort keys, then this three-level terminology is not applicable.

Note that the sort keys are written according to the desired order of the sort, and not according to the order in which the keys appear in the record. For this example, it could very well be that the three fields used as sort keys are in the following physical order in the record: CITY-NAME, STATE-NAME, COUNTY-NAME.

The USING INPUT-FILE clause in Figure 10-1 specifies the file that is the source of the record, while the GIVING SORTED-FILE clause simply specifies the file on which the sort output is to be recorded. Finally, note that in the present example the programmer does not OPEN or CLOSE any of the three files involved. The use of the SORT verb automatically takes care of such procedures.

In the above example, the whole function of the PROCEDURE DIVISION is to sort a file. This need not be the case. The SORT is simply one of the COBOL verbs and, as such, it comprises only one statement in the program. The following example illustrates the point:

```
IF  TIME-TO-SORT
    PERFORM ROUTINE-A
    SORT  CUST-SORT-FILE
          ON ASCENDING KEY NAME
       USING CUSTOMER-SOURCE-FILE
       GIVING CUSTOMER-SORTED-FILE
    PERFORM ROUTINE-B
ELSE
    PERFORM ROUTINE-C.
```

Example 2

We now illustrate use of the COBOL sort feature with a more complex data processing task. Suppose we want to read a set of records, add a field to each record to indicate its sequential order, sort the file, store the sorted file on magnetic tape, and, finally, print the sorted tape as the output of the program. Figure 10-2 presents the COBOL program designed to accomplish this task. Notice that there are four files, called INPUT-FILE, SORT-FILE, SORTED-FILE, and PRINT-FILE. DATA DIVISION entries follow the usual format, except for the use of SD to identify the SORT-FILE as the sort file, as was the case in the preceding example. In the present example, the WORKING-STORAGE SECTION is used to form the SEQUENCE-NUMBER.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. SORT2.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
*
 FILE-CONTROL.
     SELECT  INPUT-FILE ASSIGN TO READER.
*
     SELECT SORT-FILE   ASSIGN TO SORTWORK.
*
     SELECT SORTED-FILE ASSIGN TO SORTOUT.
```

FIGURE 10-2

SAMPLE SORT PROGRAM FOR EXAMPLE 2.

```
*
          SELECT  PRINT-FILE ASSIGN TO PRINTER.
*
      DATA DIVISION.
      FILE SECTION.
*
      FD  INPUT-FILE LABEL RECORD OMITTED
          DATA RECORD IS INPUT-RECORD.
      Ø1  INPUT-RECORD.
          Ø2  FILLER                   PICTURE X(1Ø).
          Ø2  NAME                     PICTURE X(15).
          Ø2  FILLER                   PICTURE X(51).
          Ø2  DATA-TO-BE-INSERTED      PICTURE 9999.
*
      SD  SORT-FILE DATA RECORD IS SORT-RECORD.
      Ø1  SORT-RECORD.
          Ø2  FILLER                   PICTURE X(1Ø).
          Ø2  NAME                     PICTURE X(15).
          Ø2  FILLER                   PICTURE X(55).
*
      FD  SORTED-FILE LABEL RECORD STANDARD
          BLOCK CONTAINS 77 RECORDS
          DATA RECORD IS SORTED-RECORD.
*
      Ø1  SORTED-RECORD                PICTURE X(8Ø).
*
      FD  PRINT-FILE LABEL RECORD OMITTED
          DATA RECORD IS PRINT-LINE.
*
      Ø1  PRINT-LINE                   PICTURE X(132).
*
      WORKING-STORAGE SECTION.
*
      Ø1  END-OF-DATA              PIC XXX.
*
      Ø1  SEQUENCE-NUMBER          PIC 9(4) VALUE ZEROS.

/
      PROCEDURE DIVISION.
*
      MAIN-SORT-ROUTINE.
          MOVE ZERO TO SEQUENCE-NUMBER.
*
          SORT SORT-FILE ASCENDING KEY NAME OF SORT-RECORD
*
               INPUT   PROCEDURE IS READING-SEQUENCING
*
               OUTPUT  PROCEDURE IS RETURNING-PRINTING.

          STOP RUN.

      READING-SEQUENCING SECTION.
*
      INPUT-SET-UP.
          OPEN INPUT INPUT-FILE.
          MOVE 'NO' TO END-OF-DATA.
*
          PERFORM READ-DATA
*
          PERFORM SEQ-RELEASE
                  UNTIL END-OF-DATA = 'YES'
          CLOSE INPUT-FILE
          GO TO END-OF-INPUT-SECTION.

      READ-DATA.
          READ INPUT-FILE RECORD
               AT END MOVE 'YES' TO END-OF-DATA.

      SEQ-RELEASE.
          ADD 1 TO SEQUENCE-NUMBER.
          MOVE SEQUENCE-NUMBER TO DATA-TO-BE-INSERTED.
```

FIGURE 10-2 (*Continued*)
SAMPLE SORT PROGRAM FOR EXAMPLE 2.

```
*
       RELEASE SORT-RECORD FROM INPUT-RECORD.
*
       PERFORM READ-DATA.

 END-OF-INPUT-SECTION.
     EXIT.

 RETURNING-PRINTING SECTION.
*
 OUTPUT-SET-UP.
     OPEN OUTPUT SORTED-FILE
                 PRINT-FILE
     MOVE 'NO' TO END-OF-DATA
*
     PERFORM RETURN-DATA
*
     PERFORM WRITE-DATA
             UNTIL END-OF-DATA = 'YES'.
     CLOSE SORTED-FILE PRINT-FILE
     GO TO END-OF-OUTPUT-SECTION.
 RETURN-DATA.
     RETURN SORT-FILE RECORD INTO SORTED-RECORD
            AT END MOVE 'YES' TO END-OF-DATA.

 WRITE-DATA.
     WRITE PRINT-LINE FROM SORTED-RECORD
     WRITE SORTED-RECORD.
*
     PERFORM RETURN-DATA.

 END-OF-OUTPUT-SECTION.
     EXIT.
```

FIGURE 10-2 *(Continued)*
SAMPLE SORT PROGRAM FOR EXAMPLE 2.

Figure 10-3 presents the program logic in flowchart form. Comparing the SORT statement in the PROCEDURE DIVISION in Figure 10-2 with the flowchart, notice that even though the SORT statement is one statement in form, it consists of three executable steps in function. These three steps are:

1 Execute the section identified by INPUT PROCEDURE IS.
2 Execute the SORT itself.
3 Execute the section identified by OUTPUT PROCEDURE IS.

In the PROCEDURE DIVISION, we first specify that we wish to sort the SORT-FILE on ASCENDING KEY NAME OF SORT-FILE. Thus, the NAME field is the sort key. INPUT PROCEDURE IS READING-SEQUENCING indicates that records will become available to the SORT-FILE according to instructions contained in a section called READING-SEQUENCING. The first paragraph in the READING-SEQUENCING SECTION, called INPUT-SET-UP, serves to open the INPUT-FILE as input. Then we enter a loop involving the SEQ-RELEASE paragraph. Each record is read, and in each case a four-digit sequence number is assigned to the field called DATA-TO-BE-INSERTED. Then we use the RELEASE SORT-RECORD FROM INPUT-RECORD statement. This simply says to move the contents of INPUT-RECORD to SORT-RECORD and then to write the SORT-RECORD on its file. The RELEASE command thus can be thought of as a specialized form of the WRITE instruction.

The loop terminates when the last record is read, at which point the program branches to END-OF-INPUT-SECTION, after INPUT-FILE is closed. The END-OF-INPUT-SECTION paragraph is the last paragraph of the READING-SEQUENCING SECTION, and is indicated by the EXIT verb. Recall that the execution of the READING-SEQUENCING SECTION was initiated by execution of the INPUT PROCEDURE statement in the SORT statement. In fact, the

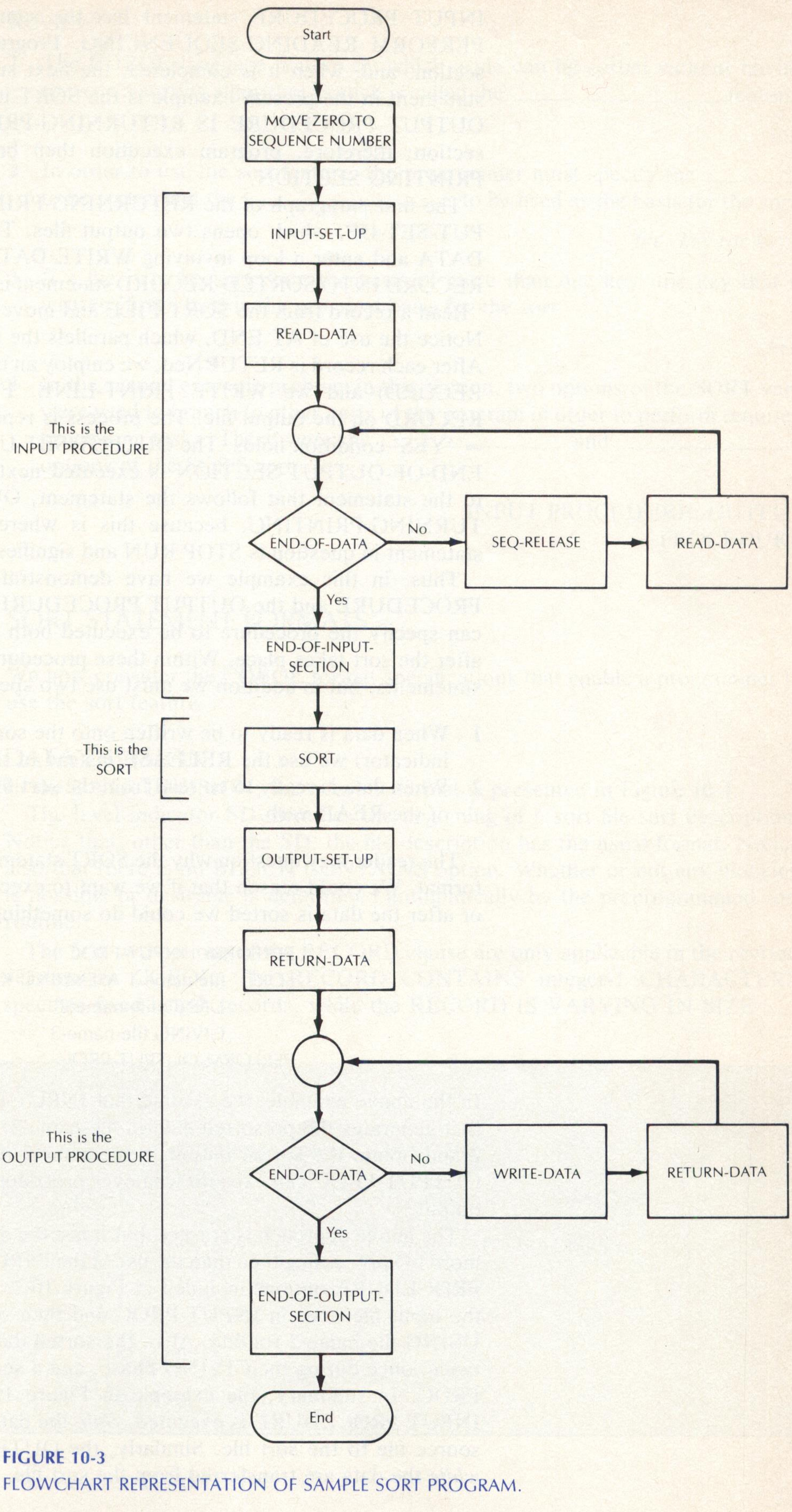


FIGURE 10-3
FLOWCHART REPRESENTATION OF SAMPLE SORT PROGRAM.

INPUT PROCEDURE statement has the same effect as if we had written PERFORM READING-SEQUENCING. Program execution branches to that section, and, when it is completed, the next statement is executed. The next statement in the present example is the SORT itself, which is then followed by OUTPUT PROCEDURE IS RETURNING-PRINTING, the name of another section; therefore, program execution then branches to the RETURNING-PRINTING SECTION.

The first paragraph of the RETURNING-PRINTING SECTION is the OUTPUT-SET-UP, which opens two output files. Then we PERFORM RETURN-DATA and enter a loop involving WRITE-DATA. The RETURN SORT-FILE RECORD INTO SORTED-RECORD statement is simply a special form of saying, "Read a record from the SORT-FILE and move it to the SORTED-RECORD." Notice the use of AT END, which parallels the same clause in the READ verb. After each record is RETURNed, we employ an implicit move (FROM SORTED-RECORD) and we WRITE PRINT-LINE. Finally, we WRITE SORTED-RECORD on the output file. The process is repeated until the END-OF-DATA = 'YES' condition holds. The OUTPUT-SET-UP paragraph closes the files and END-OF-OUTPUT-SECTION is executed next. Program control then returns to the statement that follows the statement, OUTPUT PROCEDURE IS RETURNING-PRINTING, because this is where the branching occurred. The statement in question is STOP RUN and signifies the logical end of the program.

Thus, in this example we have demonstrated that, by using the INPUT PROCEDURE and the OUTPUT PROCEDURE options of the SORT verb, we can specify the procedure to be executed both before the sort takes place and after the sort takes place. Within these procedures we can execute any COBOL statements, but in addition we must use two specialized I/O verbs:

1 When data is ready to be written onto the sort file (the one with the SD level indicator) we use the RELEASE instead of the WRITE verb.
2 When data is ready to be read from the sort file, we use the RETURN instead of the READ verb.

The reader may question why the SORT statement was written with this second format. We could reason that if we want to execute some procedure before and/or after the data is sorted we could do something like the following:

```
PERFORM   INPUT-PROC
SORT    file-name-1 ASENDING KEY key-name
        USING file-name-2
        GIVING file-name-3
PERFORM OUTPUT-PROC.
```

In the above example, we assume that INPUT-PROC executes some procedure that generates the presorted-data in file-name-2. We SORT . . . using file-name-2 and obtain the sorted output in file-name-3 (GIVING file-name-3). Then the OUTPUT-PROC can execute whatever procedure we want to do with the sorted output.

The above approach is correct, but it has the disadvantage that it will consume more I/O processing time than the use of the INPUT PROCEDURE and OUTPUT PROCEDURE options included in Figure 10-2. In the above approach we read the input file once in INPUT-PROC and then read it a second time during the USING file-name-2 routine. Also, the sorted data are read and possibly written twice, once during the GIVING phase, and a second time during the OUTPUT-PROC. In summary, the example in Figure 10-2 has the advantage that the INPUT PROCEDURE is executed *while* the data are being transferred from the source file to the sort file. Similarly, the OUTPUT PROCEDURE is executed *while* the data are transferred from the sort file.

Review

1 The COBOL language feature by which a file can be sorted without having to write a sorting algorithm as such is called the ________________ feature.

sort

2 In order to use the sort feature, the programmer must specify the ________ to be sorted and the ____________ to be used as the basis for the sort.

file; key (or keys)

3 If a file is to be sorted on the basis of more than one key, the key that is written [first / last] is the principal basis for the sort.

first

4 In the second example problem in this section, two options of the SORT verb were used to branch to other parts of the program in order to perform required processing tasks. These were the ______________ and ______________ options of the SORT verb.

INPUT PROCEDURE; OUTPUT PROCEDURE

SORT STATEMENT FORMATS

We now consider the COBOL format specifications that enable a programmer to use the sort feature.

DATA DIVISION

In the DATA DIVISION, the relevant format is presented in Figure 10-4.

The level indicator SD identifies the beginning of a sort file sort description. Notice that, other than the SD, the file description has the usual format. Notice also that there is no BLOCK CONTAINS option. Whether or not any blocking is possible or desirable is determined automatically by the preprogrammed sort routine.

The first two options of the RECORD clause are only applicable in the revised versions of COBOL. The RECORD CONTAINS integer-1 CHARACTERS specifies fixed-length records, while the RECORD IS VARYING IN SIZE . . .

```
SD   file-name-1

 [          { CONTAINS integer-1 CHARACTERS                          } ]
 [          { IS VARYING IN SIZE                                     } ]
 [  RECORD  {    [[FROM integer-2]  [TO integer-3]  CHARACTERS]      } ]
 [          {    [DEPENDING on data-name-1]                          } ]
 [          { CONTAINS [integer-4 TO] integer-5 CHARACTERS           } ]

 [        { RECORD IS   }                      ]
 [  DATA  {             }  {data-name-2} . . . ]
 [        { RECORDS ARE }                      ]
```

FIGURE 10-4
SORT FILE DESCRIPTION ENTRY.

specifies variable-length records. In the 1974 version of the language, variable-length records are specified by RECORD CONTAINS integer-4 to integer-5 CHARACTERS.

Note that in Figure 10–4 both the RECORD and the DATA clauses are optional. Usually the function of these clauses is embedded in the sort routine itself, and there is no need for the program to specify what has already been preprogrammed.

PROCEDURE DIVISION

The SORT verb is the basic verb in the SORT option. The format is presented in Figure 10–5.

The verb SORT always is required. File-name-1 is the file designated in an SD entry in the DATA DIVISION. At least one KEY has to be specified. If more than one sort key is used and all are ascending (or all descending), they can be written in the following form:

```
SORT file-name ON ASCENDING KEY ACCOUNT, NAME, YEAR.
```

Here we have specified an ascending sort by ACCOUNT, by NAME within ACCOUNT, and by YEAR within NAME. Or, we could have used the word ASCENDING (or DESCENDING) in conjunction with each KEY, as follows:

```
SORT file-name ON ASCENDING KEY ACCOUNT
               ON ASCENDING KEY NAME
               ON ASCENDING KEY YEAR.
```

The WITH DUPLICATES IN ORDER clause can be used to specify that if there are records with duplicate sort-keys, they should be kept in their original order with respect to each other. Thus if the source file contains 10 records whose name-field contains SMITH, these 10 records will be in their original order relative to each other.

The INPUT PROCEDURE and the OUTPUT PROCEDURE options refer to a section-name or a set of consecutive sections when the THRU option is used. The paragraphs in such sections specify the processing tasks to be performed prior to the sort (INPUT PROCEDURE) or after the sort (OUTPUT PROCEDURE). If the INPUT PROCEDURE is used, the verb RELEASE must be used somewhere in that procedure. If the OUTPUT PROCEDURE is used, the verb RETURN must be used somewhere in that procedure. The USING file-name 2 option is used when records are made available to the sort from file-name-2 without any processing. The GIVING file-name-3 option specifies that the sorted file is to be recorded on file-name-3.

```
SORT file-name-1

{ON {ASCENDING } KEY {data-name-1} . . . } . . .
    {DESCENDING}

[COLLATING SEQUENCE IS alphabet-name-1]
[WITH DUPLICATES IN ORDER]

{INPUT PROCEDURE IS {section-name-1  } [{THROUGH} {section-name-2  }] }
{                   {procedure-name-1} [{THRU   } {procedure-name-2}] }
{USING {file-name-2} . . .                                             }

{OUTPUT PROCEDURE IS {section-name-3  } [{THROUGH} {section-name-4  }] }
{                    {procedure-name-3} [{THRU   } {procedure-name-4}] }
{GIVING {file-name-3} . . .                                             }
```

FIGURE 10-5
THE SORT STATEMENT.

```
RETURN file-name-1 RECORD [INTO identifier]
  AT END imperative-statement-1
  [NOT AT END imperative-statement-2]
  [END-RETURN]
```

FIGURE 10-6
THE RETURN STATEMENT.

In the revised version of COBOL, INPUT PROCEDURE and OUTPUT PROCEDURE need not refer to section-names. They can be either paragraphs or sections, thus allowing for greater flexibility. In the example in Figure 10–2 we used the section choice, and as a result we had to use the GO TO verb to reach the end-paragraph in the section. The revised version allows us to eliminate the END-OF-INPUT-SECTION and the END-OF-OUTPUT-SECTION paragraphs as well as the two GO TO statements in Figure 10–2.

```
RELEASE record-name [FROM identifier].
```

The RELEASE verb can be used only in a section referenced by the INPUT PROCEDURE. The record-name in this format refers to a record in the sort file. If the FROM option is used, the effect is to move the contents of identifier to the record-name and then to RELEASE. In effect, RELEASE is a specialized form of the WRITE verb.

The RETURN verb, which is used in conjunction with the OUTPUT PROCEDURE of a SORT verb, has the format presented in Figure 10-6.

The RETURN verb has the effect of a READ verb. The file-name is the name of the sort file. When the INTO option is used, the effect is the same as execution of the two statements RETURN file-name MOVE record-name TO identifier. The AT END clause is required.

Notice that in the revised version we can use the NOT AT END and the END-RETURN options in the RETURN statement for handling conditionals more easily. For example, we could modify the RETURNING-PRINTING-SECTION of Figure 10-2 as follows (including omission of the SECTION requirement).

```
RETURNING-PRINTING.
    OPEN  OUTPUT  SORTED-FILE
                  PRINT-FILE
    MOVE 'NO' TO END-OF-DATA
    PERFORM RETURN-DATA
        UNTIL END OF DATA = 'YES'.
    CLOSE SORTED-FILE
          PRINT-FILE.
RETURN-DATA.
    RETURN SORT-FILE RECORD INTO SORTED-RECORD
        AT END
            MOVE 'YES' TO END-OF-DATA
        NOT AT END
            WRITE PRINT-LINE FROM SORTED-RECORD
            WRITE SORTED-RECORD
    END-RETURN.
```

In the above example the END-RETURN is not necessary. However, if we wanted to execute a statement regardless of the outcome of the AT END condition test, we would need to use the END-RETURN scope terminator so that the statement(s) that follow the END-RETURN are executed unconditionally.

Review

1 In the DATA DIVISION, the file to be sorted is identified by the level indicator ________________.

SD (*standing for Sort Description*)

2 The option of specifying a variable-length record depending on a data-name in the sort description is available only in the [1974 version / revised version] of COBOL.

revised version

3 If the INPUT PROCEDURE option is used in conjunction with the SORT verb, designated processing is performed [before / after] the sort, and the verb ________________ must be used somewhere in the procedure.

before; RELEASE

4 If the OUTPUT PROCEDURE option is used in conjunction with the SORT verb, designated processing is performed [before / after] the sort, and the verb ________________ must be used somewhere in the procedure.

after; RETURN

5 The RELEASE verb can be considered a specialized form of the __________ verb, while the RETURN verb can be considered a form of the __________ verb.

WRITE; READ

6 In order to handle conditionals more easily, the NOT AT END and the END-RETURN options can be used in a RETURN statement that utilizes the [1974 version / revised version] of COBOL.

revised version

FILE MERGING

A very common data processing procedure is that of merging two or more files. Merging simply means combining two or more files into one file. For example, a bank may create a tape file containing the day's transactions in customer number order on a daily basis. Then, end-of-month processing requires that the daily tapes be combined into one, so that the entire month's transactions for each customer are accumulated.

Because it is frequently needed and of great importance, merging is implemented in COBOL as a very high-level language feature, in the form of the MERGE statement.

Let us consider an example. A business firm generates a sales history file at the end of the quarter. Each record in the file contains a department number and a product number, as well as many other fields. This quarterly file is sorted, with department number being the major sort key and product number being the minor sort key. At the end of the year we are interested in merging the four quarterly sales history files into one. Figure 10-7 presents an outline of the relevant parts of the program. Four files are introduced with an FD entry, one for each quarter. The fifth FD entry is for the output file. Then the SD introduces the file to be used for the merge, which in this example is called MERGE-FILE. Notice that the data record description for this file corresponds to the record description of the four quarterly files. The merge statement in the PROCEDURE DIVISION references the SD file and specifies that the merge will proceed on the basis of DEPARTMENT being the major key and PRODUCT being the minor key. As

```
FD   FIRST-QUARTER LABEL RECORDS STANDARD
                         DATA RECORD  SALES-HISTORY.
01   SALES-HISTORY.
     02  DEPT-NO        PIC 999.
     02  PROD-NO        PIC 99999.
     .
     .
     .
FD   SECOND-QUARTER . . .
     .
     .
     .
FD   THIRD-QUARTER . . .
     .
     .
     .
FD   FOURTH-QUARTER . . .
     .
     .
     .
FD   YEARLY LABEL RECORDS STANDARD
                    DATA RECORD  CUMULATIVE-SALES.
01   CUMULATIVE-SALES.
     02  DEPT-NO        PIC 999.
     02  PROD-NO        PIC 99999.
     .
     .
     .
SD   MERGE-FILE DATA RECORD  MERGE-RECORD.
01   MERGE-RECORD.
     02  DEPARTMENT         PIC 999.
     02  PRODUCT            PIC 99999.
     .
     .
     .
PROCEDURE DIVISION.
     .
     .
     .
     MERGE MERGE-FILE ON ASCENDING KEY DEPARTMENT
                      ON ASCENDING KEY PRODUCT
          USING FIRST-QUARTER, SECOND-QUARTER,
                THIRD-QUARTER, FOURTH-QUARTER
          GIVING YEARLY.
```

FIGURE 10-7
OUTLINE FOR A MERGE PROGRAM.

is the case with the SORT verb, the keys decrease in significance in the order written. The ASCENDING option specifies that the next record of each of the four quarterly files will be examined, and the record sent to the output file next is the one that has the highest department number, or the highest product number if the department numbers are equal. If all four records have identical department and product values, then the records will be sent to the output file in the order in which the file names are written in the merge statement.

```
MERGE file-name-1
 { ON  { ASCENDING  }  KEY   {data-name-1} . . . } . . .
       { DESCENDING }
 [COLLATING SEQUENCE IS alphabet-name-1]
 USING file-name-2 {file-name-3} . . .
 { OUTPUT PROCEDURE IS { section-name-1   } [ { THROUGH } { section-name-2   } ] }
 {                     { procedure-name-1 } [ { THRU    } { procedure-name-2 } ] }
 { GIVING    {file-name-4} . . .                                                  }
```

FIGURE 10-8
THE MERGE STATEMENT.

The USING clause specifies the files to be merged, which are the input files. These files must be closed at the time of merging. Opening is carried out by the MERGE statement in an implicit fashion.

The GIVING clause specifies the output file. This file will contain the combined set of the four quarterly files. This new file will be in the same sort order as the quarterly files. Note that in order for the merge process to take place correctly the input files must be in the sort order indicated by the KEY specifications.

The general format of the MERGE statement is presented in Figure 10-8.

The OUTPUT PROCEDURE option parallels the one available with the SORT verb. A RETURN statement is used within the output procedure to make merged records available for processing, just as is the case with SORT. Unlike SORT, MERGE does not include any input procedure options; thus, the input files must be in proper form for merging before a MERGE instruction is executed.

Review

1 The COBOL language feature by which monthly summaries of transactions can be combined to create an annual summary is the ________________ statement.

MERGE

2 If 12 monthly summaries are to be combined to form an annual summary, then the number of FD entries required in the associated MERGE program is ________________ (number).

13

3 In order for the merge process to take place correctly, it [is / is not] necessary that each input file be in the exact sort order indicated by the KEY specifications.

is

EXERCISES

10.1 A sort file has been defined as SORT-FILE, and, in part, its data division entries include:

```
02  COURSE-CODE      PIC XXX.
02  COLLEGE          PIC 99.
02  COURSE-NAME      PIC X(6).
```

Using the following data, write a SORT statement that could cause the sorted data shown. The original data come from SOURCE-FILE, and we

want to have the sorted data in SORTED-FILE. Be sure to specify which are the major, intermediate, and minor sort keys.

ORIGINAL DATA	SORTED DATA
CIS20BILL	MGT10JILL
CIS30LINDA	QBA10BRENDA
QBA10BRENDA	CIS20MARY
CIS30XAVIER	CIS20JOHN
MGT10JILL	CIS20BILL
CIS20JOHN	CIS30XAVIER
CIS20MARY	CIS30LINDA

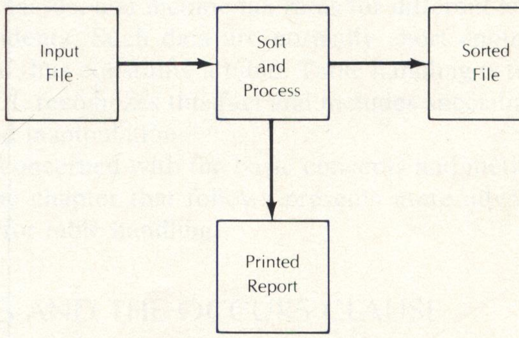
10.2 A file contains data about students and has the following record format:

FIRST-NAME	PIC	X(10).
LAST-NAME	PIC	X(15).
YEAR	PIC	X(2).
MAJOR	PIC	X(3).
GPA	PIC	9V99.

Write a program to sort the file so that student records are in order by year of studies (YEAR) within major field of study (MAJOR) and in descending order of GPA. In addition, the sorted file must have a different format from the original file: the FIRST-NAME and LAST-NAME fields must be reversed:

LAST-NAME	PIC	X(15).
FIRST-NAME	PIC	X(10).
YEAR	PIC	X(2).
MAJOR	PIC	X(3).
GPA	PIC	9V99.

The sorted file is to be saved as a separate file as shown below:



It is also desired to produce a report from the newly sorted file as presented in Figure 10-9. Use the sample data given in Figure 10-10 as input for your program.

10.3 Using any data file available, write a program incorporating the COBOL sort feature to sort a file. For example, you could modify any of the exercises at the end of Chapter 9 to sort the master file or the transaction file in the required order.

10.4 Consider the following to be the contents of the four quarterly files discussed in the merging example in this chapter. Show the content of the output file.

FIRST QUARTER	SECOND QUARTER	THIRD QUARTER	FOURTH QUARTER
345 12345	123 00112	345 56111	931 00001
345 25936	987 56111		999 99999
619 01110			

10.5 Using any two sorted data files, write a program incorporating the MERGE feature to combine the two files into one file.

```
MAJOR FIELD: XXX [New page for each new field]

STUDENT NAME     YEAR     GPA     AVG. GPA
  L1  F1         FR       3.40
  L2  F2         FR       2.00
  L3  F3         FR       1.90
                                   2.43
  L4  F4         SO
                  )
                 JR
                  )
                 SR
                  )
      AVG. GPA FOR XXX MAJOR FIELD = 9.99
```

FIGURE 10-9
REPORT FORMAT FOR EXERCISE 10.2.

MAURICE	HOLLMAN	03CIS273
ADAM	LENHARDT	01ACC205
ROBERT	HAYWOOD	04MKT312
ALLAN	TEEGARDEN	05MGT400
JAMES	NORVELL	02FIN302
ROGER	WHITTIER	02FIN395
MARGARET	AKIN	05ACC295
CHRIS	NORTON	01ACC271
JIM	BEECHER	01CIS240
TOD	FORBES	03MGT362
FLOYD	MCNEELY	04ACC314
MICHAEL	DERKS	04MGT268
DEAN	WERNER	01MKT250
CORDELIA	MONTGOMERY	04MKT400
MAGGIE	WILSON	03FIN254
JACK	HOLT	02ACC267
LEONARD	LESMEISTER	02MGT332
DONALD	FAUBERT	04ACC348
WILLARD	FICKER	01FIN257
EDWARD	EASTON	03MKT260
MICHAEL	VOLRICH	02CIS378
RITA	SOLANO	02ACC400
BRYON	ELLIS	01FIN285
FRANCIS	QUIGLEY	04CIS395
CORNELIUS	CLAXTON	03FIN215
HERBERT	SCHAEFER	03MGT400
ROSELLA	MCGOWEN	04MKT390
LAURA	HOFSTATTER	03CIS268
GORDON	PETRIE	01FIN240
TRACY	ZIMMERMAN	03ACC272
EVELYN	RAGSDALE	02FIN278
SIDNEY	KRAMER	01MKT179
GIBSON	GORMAN	02MKT349
LINDSEY	YOUNGBLOOD	03MGT311
STANLEY	FORRESTER	04CIS329
ROBERT	UPDIKE	04FIN268
LESTER	CROWLEY	03ACC287
ELWOOD	ISAAC	01FIN305
MORRIS	JACOBY	02CIS298
JESSIE	LANGFORD	04MGT205
MERRILL	ORMSBEE	03CIS400

FIGURE 10-10
SAMPLE INPUT DATA FOR EXERCISE 10.2.

11 BASIC TABLE HANDLING

■ INTRODUCTION

A table, like a file, is a collection of logically related entries. Examples are tax rates for different municipalities in a metropolitan area, commission rates for different product classes, and income tax rates for different levels of income and numbers of dependents. Such data are normally short enough to be placed in central storage and thus constitute a table. Table handling is fundamental to data processing. COBOL recognizes this fact and includes specialized instructions for table definition and manipulation.

This chapter is concerned with the basic concepts and methods for processing tables of data. The chapter that follows presents more advanced methods and language features for table handling.

■ SUBSCRIPTING AND THE OCCURS CLAUSE

A great deal of the documentation in COBOL derives from the use of appropriate data-names, that is, names that provide a direct clue to the type of data contained in the named storage location. There are situations, however, when practicality dictates that we dispense with the use of such names. For example, suppose we are processing data on the average income per household in each of the 50 states. If we chose to name the average income for each state uniquely, we could have such data names as ALABAMA-INCOME, ALASKA-INCOME, and so on, for a total of 50 names. It is easy to imagine the problems that this practice would cause in the PROCEDURE DIVISION. For example, 50 MOVE statements would be required before the results could be printed.

The use of tables and subscripts is a programming feature that is particularly useful in such situations. A table is simply a set of values stored in consecutive

storage locations and assigned one data-name. Reference to specific entries in the table is made by the use of the one name along with a subscript that identifies the location of the particular entry. Entries in a one-dimensional table are numbered sequentially 1, 2, 3, . . . , on to the last. Thus, in our example of the average household income for the 50 states, imagine that we have a table of 50 entries. If the entries are arranged alphabetically and we wish to reference the average income for Arizona, the subscript will have a value of 3. Similarly, the subscripts for Washington and Wyoming will be 49 and 50, respectively. Use of the OCCURS clause in conjunction with the PICTURE clause enables the programmer to set up tables so that reference can be made to entire tables or individual values in tables by means of subscripts. A DATA DIVISION entry involving an OCCURS clause includes the data-name assigned to the table, the number of dimensions, the number of entries in each dimension, and the field characteristics of the entries. In this section, we consider one-dimensional tables only, such as the one for the average household income in the 50 states; in the next section we discuss two- and three-dimensional tables.

Assume that the data for average income is contained in a WORKING-STORAGE table, although it could be a FILE SECTION table just as well. Thus, we have:

```
WORKING-STORAGE SECTION.
01  . . .
01  . . .
01  STATE-INCOME-TABLE.
    02  AVERAGE-INCOME OCCURS 50 TIMES      PICTURE 9(6)V99.
```

The OCCURS 50 TIMES clause sets up a table in storage that has the conceptual structure portrayed in Figure 11-1.

Execution of the PROCEDURE DIVISION statement MOVE STATE-INCOME-TABLE . . . will result in the entire table of 50 fields being moved. In order to move (or otherwise process) a single field or entry in the table, the subscript is included in parentheses and separated from the name by a space, as follows: MOVE AVERAGE-INCOME (12) TO . . . This statement, of course, refers to the twelfth table entry.

The subscript may be a variable instead of a constant, but it always must be a positive integer (whole number). To understand the need for a subscript that is a variable, consider the following example. Suppose that some records have the following layout:

```
01  STAT-REC.
    02  STATE-NUMBER        PICTURE 99.
    02  INCOME              PICTURE 9(6)V99.
```

Thus, the value in columns 1–2 is the number of the state when the states are listed alphabetically. The value in columns 3–10 is the average household income for that state. If the records have not been arranged alphabetically, the following

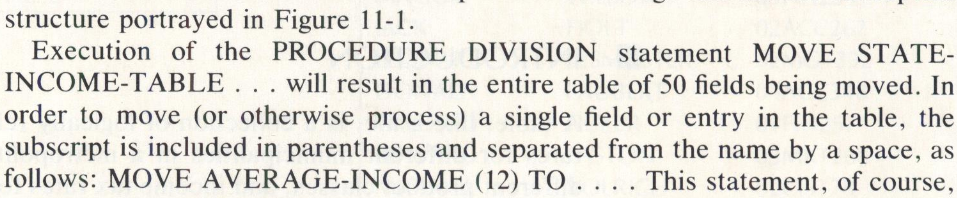


FIGURE 11-1

CONCEPTUAL STRUCTURE OF A TABLE OF AVERAGE INCOME BY STATE.

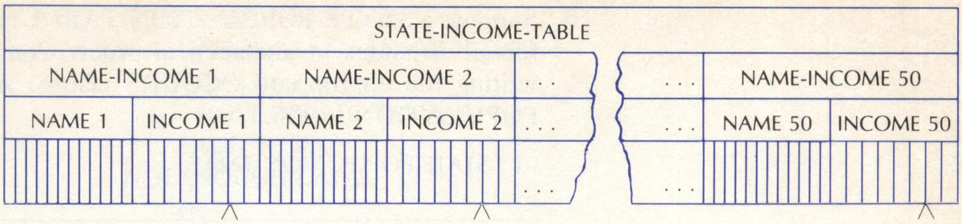


FIGURE 11-2
STRUCTURE OF A TABLE OF AVERAGE INCOME THAT INCLUDES STATE NAMES.

statement can be used to insert the income figure in the appropriate place in the table after the record has been read:

```
MOVE INCOME TO AVERAGE-INCOME (STATE-NUMBER).
```

As a result of this statement, if the state number were 49, the income value would be inserted in the forty-ninth entry of the average income table.

The OCCURS clause need not be used alone in a record; other reference entries may be included as well. For example, the record might have been structured as follows:

```
01  STATE-INCOME-TABLE.
    02  AVERAGE-INCOME OCCURS 50 TIMES     PICTURE 9(6)V99.
    02  NATIONAL-AVERAGE                   PICTURE 9(6)V99.
```

Notice, however, that STATE-INCOME-TABLE now refers to more than the table of 50 entries. If we want to make specific reference to the table of 50 entries, we will have to write something like this:

```
01  STATE-INCOME-TABLE.
    02  AV-TABLE.
        03  AVERAGE-INCOME OCCURS 50 TIMES     PICTURE 9(6)V99.
    02  NATIONAL-AVERAGE                       PICTURE 9(6)V99.
```

As a further illustration of a one-dimensional table, assume that we want to include the names of the states along with their corresponding average income figures:

```
01  STATE-INCOME-TABLE.
    02  NAME-INCOME OCCURS 50 TIMES.
        03  NAME      PICTURE X(12).
        03  INCOME    PICTURE 9(6)V99.
```

The OCCURS 50 TIMES clause sets up a table in storage that has the structure portrayed in Figure 11-2.

If we write NAME (1) we are referring to a storage field of 12 positions, whereas INCOME (1) refers to an 8-position field. If we write NAME-INCOME (1) we are referring to a storage field of 20 positions. Finally, STATE-INCOME-TABLE refers to the entire table of 100 fields.

Review

1 The programmer can set up tables by using the ________________ clause in the DATA DIVISION.

OCCURS

2 The OCCURS clause indicates the number of ________________ in the table.

entries

3 Suppose a STATE-POPULATION-TABLE is to include the population figures for all 50 states in alphabetical order. Complete the description below by writing the appropriate OCCURS clause. Assume that the PICTURE for POPULATION is 9(8).

```
01 STATE-POPULATION-TABLE.
```

____

02 POPULATION OCCURS 50 TIMES PICTURE 9(8).

4 Suppose that both the state names and the population figures are read in and we wish to set up a STATE-POPULATION-TABLE such that the 50 state names are located first in the table, followed by the 50 population figures. Complete the following description, assuming that the PICTURE for NAME is X(12).

```
01 STATE-POPULATION-TABLE.
```

____

____

02 NAME OCCURS 50 TIMES PICTURE X(12).
02 POPULATION OCCURS 50 TIMES PICTURE 9(8).

5 The table set up in the preceding question will have a total of 100 fields. After all data are read in, the content of NAME (1) will be the state name ____________, and the content of POPULATION (50) will be the population figure for the state of ____________.

Alabama; Wyoming

READING VALUES INTO A TABLE

Let us suppose that we have defined a tax table to contain 10 deduction rates, thus:

```
01 TAX-TABLE.
   02 TAX-RATE OCCURS 10 TIMES      PICTURE V999.
```

We want to read in 10 values from a source file. Assume that the source file is called RATE-FILE, and that the specific field containing the rate is called RATE. We will use a data-name, N, to specify the subscript value. Initially, we want to read the first record and store the value of RATE in the first cell of the TAX-TABLE. Then we want to increase the value of N and repeat the process, storing each newly read value in the Nth place of TAX-TABLE. The following PROCEDURE DIVISION entries can accomplish the rate-reading objective. Notice that we account for the possibility of less than 10 records in the input file, in which case we execute an error routine called NOT-ENOUGH-DATA.

```
MOVE 'NO' TO DATA-END
MOVE 1 TO N
PERFORM TABLE-READ UNTIL N > 10 OR DATA-END = 'YES'.
.
.
.
TABLE-READ.
    READ RATE-FILE RECORD AT END MOVE 'YES' TO DATA-END.
```

```
      IF DATA-END = 'YES'
          PERFORM NOT-ENOUGH-DATA
      ELSE
          MOVE RATE TO TAX-RATE (N)
          ADD 1 TO N.
  NOT-ENOUGH-DATA.
      (etc.)
```

ENTERING CONSTANT VALUES INTO A TABLE

It is often desirable to build tables that contain specified constant values. One way to accomplish this objective is to define the table by using the OCCURS clause in the DATA DIVISION and then to read in the desired values through suitable PROCEDURE DIVISION instructions. This approach generally is used when there are a large number of constant values to be entered in the table. But when the number of constant values is limited, other approaches, as described in this section, are commonly used.

Use of the REDEFINES Clause

Suppose that we want to have a table that contains the names of the 12 months of the year, so that we can reference these names by use of the table name and a subscript. For instance, we may want to reference the fifth month or the twelfth month, and so on. Using numeric values to reference the months is desirable, because arithmetic can be performed with numeric values. For instance, if we are on the sixth month and we want to reference the next month, we can simply add 1 to 6 and then make reference to the resulting month. The following example illustrates the common way of accomplishing this task:

```
01  MONTH-TABLE.
    02  FILLER     PICTURE X(9)  VALUE 'JANUARY  '.
    02  FILLER     PICTURE X(9)  VALUE 'FEBRUARY '.
    02  FILLER     PICTURE X(9)  VALUE 'MARCH    '.
    02  FILLER     PICTURE X(9)  VALUE 'APRIL    '.
    02  FILLER     PICTURE X(9)  VALUE 'MAY      '.
    02  FILLER     PICTURE X(9)  VALUE 'JUNE     '.
    02  FILLER     PICTURE X(9)  VALUE 'JULY     '.
    02  FILLER     PICTURE X(9)  VALUE 'AUGUST   '.
    02  FILLER     PICTURE X(9)  VALUE 'SEPTEMBER'.
    02  FILLER     PICTURE X(9)  VALUE 'OCTOBER  '.
    02  FILLER     PICTURE X(9)  VALUE 'NOVEMBER '.
    02  FILLER     PICTURE X(9)  VALUE 'DECEMBER '.
01  MONTHS REDEFINES MONTH-TABLE.
    02  MONTH     PICTURE X(9) OCCURS 12 TIMES.
```

Notice that the record MONTH-TABLE consists of 12 fields with each field containing the name of a month. The VALUE clause is used to assign the constant (nonnumeric literal) values. The record called MONTHS is a table consisting of 12 entries. Each entry is referenced by the use of MONTH and a subscript. Thus, executing the instruction MOVE MONTH (3) TO PRINTAREA WRITE PRINTAREA results in the word MARCH being printed.

A practical example is given later in this chapter, using the table of months.

The procedure may seem unnecessarily roundabout. Let us justify the rationale. In the 1974 version of COBOL the VALUE clause cannot be used with the OCCURS clause. This makes sense, since the VALUE clause references *one* value and the OCCURS clause refers to several values. Thus, in the example

above we use the REDEFINES clause with the OCCURS clause, after we have described each individual field. Of course, all entries in MONTH-TABLE must be of equal field size for the procedure to accomplish the correct result.

Use of the VALUE Clause

Unlike the 1974 version of COBOL, the revised version permits use of the combination of the OCCURS and VALUE clauses to enter the same constant into all of the positions in a table. For example, we can write:

```
01  SAMPLE-TABLE.
    02  TABLE-CELL OCCURS 100 TIMES
                   PIC 9(5)V99
                   VALUE ZERO.
```

When the above program segment is executed, each of the 100 fields in the table is set to zero.

Another approach that is available in both the 1974 and the revised versions of COBOL permits the entry of different constants into the different table positions. The technique involves use of the VALUE clause at a level superior to the OCCURS clause, as illustrated in the following example:

```
01  ALPHABET-TABLE
        VALUE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'.
    02  LETTER OCCURS 26 TIMES PIC X.
```

By using the VALUE clause at the 01 level in the example above, we built a table of 26 cells, each cell containing a different letter of the alphabet.

Use of the MOVE Verb

Another way of entering either the same or different constants in the various positions of a table is to reference the superior level of the OCCURS clause in an appropriate MOVE statement. For example we could write:

```
01  SAMPLE-TABLE.
    02  TABLE-CELL OCCURS 100 TIMES
                   PIC 9(5)V99.
        .
        .
        .
    MOVE ZEROS TO SAMPLE-TABLE.
```

The MOVE statement fills the 700-byte SAMPLE TABLE field with zeros. As a result, each of the 7-byte TABLE-CELL fields will also contain zero values. However, this approach is appropriate only when numeric fields have USAGE DISPLAY, either explicitly, or implicitly, as above. As we discussed in Chapter 8, we may also use USAGE COMP for numeric fields. If so, we have to be aware that when dealing with group items, zero data is different from zero data in elementary USAGE COMP items. To illustrate the point, consider this example:

```
01  SAMPLE-TABLE.
    02  TABLE-CELL  OCCURS 100 TIMES
                    PIC 9(5)V99
                    USAGE COMP.
        .
        .
        .
    MOVE ZEROS TO SAMPLE-TABLE.
```

The MOVE statement fills SAMPLE-TABLE with zero characters. However, USAGE COMP specifies noncharacter representation for the data in each TABLE-CELL. Should we attempt to do arithmetic using TABLE-CELL subsequent to the MOVE, the results are unpredictable.

As a final example of moving data to the group level, we could rewrite the alphabet example as:

```
01  ALPHABET-TABLE.
    02  LETTER OCCURS 26 TIMES PIC X.
    .
    .
    .
    MOVE 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' TO ALPHABET-TABLE.
```

As a result of executing the program segment above, each LETTER will contain one of the letters of the alphabet.

Review

1 One way of establishing a table of constant values without reading them in through PROCEDURE DIVISION statements is through use of the ______ clause in conjunction with the OCCURS option.

REDEFINES

2 Another approach, by which only the same constant can be entered in all positions of the table, concerns use of the combination of the OCCURS and VALUE clauses. This approach is available only with the [1974 standard / revised version] of COBOL.

revised version

3 Another way available with both versions of COBOL that the VALUE clause can be used, and which permits entry of different constants into different table positions, is to use the VALUE clause at a level [superior / subordinate] to the OCCURS clause.

superior

4 The third approach for entering constants in a table, and by which either the same or different constants can be entered, is by use of the MOVE verb. However, the MOVE statement will have the intended result only if the numeric fields have been described either explicitly or implicitly as being USAGE ______.

DISPLAY

THE OCCURS . . . DEPENDING ON OPTION

Sometimes the number of entries in a table varies. The number of entries may be given by the value of a data-name. In such cases we may want to use the DEPENDING ON option of the OCCURS clause. In Chapter 9 we mentioned variable-sized file records and deferred discussion until this point. We now illustrate the use of variable file records with the OCCURS and DEPENDING ON clauses.

A bank utilizes magnetic tape to record the transactions of checking account customers. Some customers have a greater number of transactions than others; that is, they write more checks or make more deposits. It seems natural that tape records should be variable. Let us assume the record layout presented in Figure 11-3.

FIELD	NUMBER OF POSITIONS
Customer number	6
Number of transactions	3
Transaction code	1
Date	5
Amount	7
Transaction code	1
Date	5
Amount	7
(etc. for up to 100 transactions)	

FIGURE 11-3
RECORD LAYOUT FOR CHECKING ACCOUNT CUSTOMERS.

This is a case where a record may contain from 0 to 100 transactions. Notice that the minimum number of character positions is nine: six for the customer number and three for the number of transactions. The maximum size is 9 + (100 transactions × 13 characters per transaction) = 1,309. We then can have the following file description, assuming blocks of three records each:

```
FD  TAPE-FILE          BLOCK CONTAINS 3 RECORDS
                       RECORD CONTAINS 9 TO 1309 CHARACTERS
                       LABEL RECORD STANDARD
                       DATA RECORD IS CHECKING-ACCOUNT-RECORD.
01  CHECKING-ACCOUNT-RECORD.
    02  CUSTOMER-NUMBER              PICTURE 9(6).
    02  NUMBER-OF-TRANSACTIONS       PICTURE 999.
    02  TRANSACTION OCCURS 1 TO 100  TIMES DEPENDING ON
        NUMBER-OF-TRANSACTIONS.
        03  TRANSACTION-CODE         PICTURE 9.
        03  TRANSACTION-DATE         PICTURE 9(5).
        03  TRANSACTION-AMOUNT       PICTURE 9(5)V99.
```

Some comments are in order. First, the statement RECORD CONTAINS 9 TO 1309 CHARACTERS is optional, since the record description provides the same information. It should be pointed out that NUMBER-OF-TRANSACTIONS does not automatically contain the number of transactions. It is the responsibility of the program logic to store the proper data in the data-name of the DEPENDING-ON clause.

The general form of the DEPENDING ON option is

```
. . . OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name.
```

In the 1974 standard version of COBOL, integer-1 must be 1 or greater, as indicated in the above format. In the revised version, however, integer-1 may be zero.

Review

1 The OCCURS . . . DEPENDING ON option can be used when the number of entries to be included in a table is [predetermined / variable].

variable

2 When the DEPENDING ON option is used, the word OCCURS in the program statement always is followed by a specified [value / range of values], and the

phrase DEPENDING ON always is followed by a [data-name / specified value].

range of values; data-name

THE PERFORM VERB AND TABLE HANDLING

The PERFORM verb was introduced in Chapter 2, and its use was described further in Chapter 5. We now continue our study of the PERFORM verb by introducing additional formats and emphasizing the use of this verb for table handling applications.

Beginning with an example, suppose that we have monthly sales for the 12 months of the year, and we wish to compute the average monthly sales. The data has been stored in SALES-TABLE as follows:

```
01  SALES-TABLE.
    02  MONTHLY-SALES       PIC 9(6)V99 OCCURS 12 TIMES.
```

To compute the average monthly sales we can write:

```
    MOVE ZERO TO TOTAL-SALES
    MOVE 1 TO N
    PERFORM SUMMATION 12 TIMES
    DIVIDE TOTAL-SALES BY 12 GIVING AVERAGE-SALES.
    .
    .
    .
SUMMATION.
    ADD MONTHLY-SALES (N) TO TOTAL-SALES
    ADD 1 TO N.
```

Instead of an explicit reference, such as PERFORM . . . 12 TIMES, we can use an identifier whose value is subject to change. For instance, the previous example could be modified by using K as the identifier that contains the number of months for which we want to compute a sales average:

```
    MOVE ZERO TO TOTAL-SALES
    MOVE 1 TO N
    PERFORM SUMMATION K TIMES
DIVIDE TOTAL-SALES BY K GIVING AVERAGE-SALES.
    .
    .
    .
SUMMATION.
    ADD MONTHLY-SALES (N) TO TOTAL-SALES
    ADD 1 TO N.
```

Note, however, that if the integer or the identifier used with PERFORM has a value of zero or is negative, the effect is that the object of PERFORM is not executed. For example, if we say PERFORM ABC M TIMES and M happens to have a value of 0 or is negative, ABC will not be executed at all; rather, the program will go on to the next statement. In effect, this fact allows for conditional execution of a procedure, but in a confusing, implicit way.

There is available a format of the PERFORM verb that provides a convenient way of processing data in tables. This format is presented in Figure 11-4.

It will be easier to understand the components of this version of PERFORM

PERFORM procedure-name-1 [THRU procedure-name-2] VARYING identifier-1 FROM {identifier-2 / literal-1} BY {identifier-3 / literal-2} UNTIL condition

FIGURE 11-4

FORMAT OF THE PERFORM VERB WITH THE VARYING OPTION.

if we first consider an example. Let us take the example that we have been describing and rewrite those instructions using the alternative format:

```
MOVE ZERO TO TOTAL-SALES
PERFORM SUMMATION VARYING N FROM 1 BY 1
    UNTIL N > 12
DIVIDE TOTAL-SALES BY 12 GIVING AVERAGE-SALES.
.
.
.
SUMMATION.
    ADD MONTHLY-SALES (N) TO TOTAL-SALES.
```

The use of PERFORM . . . VARYING allows us to execute an object paragraph or paragraphs while systematically varying an identifier. Of course, this identifier (in the above example, N) must have been defined in the DATA DIVISION. Most often, the identifier varied is used as a subscript, as in this example; however, it could be used simply as a counter to control the number of executions of the object of the PERFORM verb. The flowchart in Figure 11-5 portrays the control logic involved in the execution of PERFORM with the VARYING option.

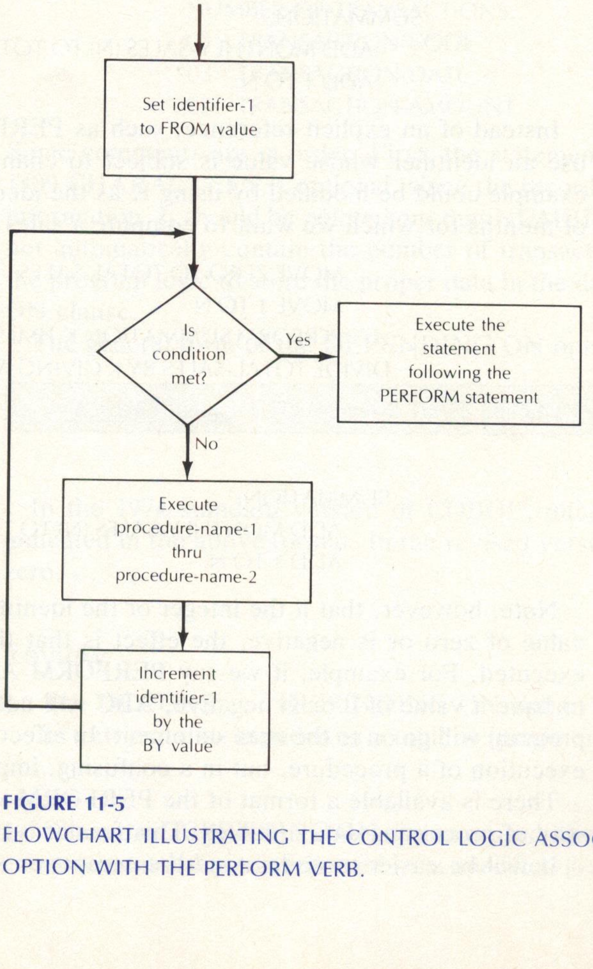


FIGURE 11-5

FLOWCHART ILLUSTRATING THE CONTROL LOGIC ASSOCIATED WITH USING THE VARYING OPTION WITH THE PERFORM VERB.

It is worthwhile to consider some further examples in order to illustrate the potential of PERFORM . . . VARYING. Let us say a home mortgage company issues a set of payment coupons at the beginning of each year. There are 12 coupons, numbered 1 to 12, each containing the name of the month in which the payment is due and the amount. What is required in order to prepare the coupons, then, is a repetitive execution (12 times) of a task with two variable factors: the coupon number (01 to 12) and the month-name. Of course, the amount of payment due will be the same for each of the months, and our present example is not concerned with the determination of this amount.

First, let us set up partial DATA DIVISION entries:

```
01  MONTH-NUMBER            PICTURE 99.
01  MONTHS-TABLE.
    02  MONTH               PICTURE X(9) OCCURS 12 TIMES.
01  COUPON. (fillers skipped)
    02  COUPON-NUMBER       PICTURE 99.
    02  MONTH-NAME          PICTURE X(9).
    02  EDITED-AMOUNT       PICTURE $$,$$9.99.
```

The MONTHS-TABLE will be filled with the names of the 12 months. Assume that the names are to be read from a file, the first record containing the name JANUARY and the twelfth record containing the name DECEMBER, in a field called REC-MONTH. The following PROCEDURE DIVISION program segment can be used to accomplish this task:

```
PERFORM MONTH-READING
    VARYING MONTH-NUMBER FROM 1 BY 1
    UNTIL MONTH-NUMBER > 12.

MONTH-READING.
    READ MONTH-FILE RECORD
        AT END MOVE 'YES' TO END-OF-DATA.
    IF END-OF-DATA = 'YES'
        NEXT SENTENCE
    ELSE
        MOVE REC-MONTH TO MONTH (MONTH-NUMBER).
```

Once the 12 month-names have been entered in the MONTHS-TABLE by execution of the above program segment, the set of statements required to print the 12 coupons, each with a coupon number, month-name, and edited amount, can be written as:

```
PERFORM COUPON-PRINTING
    VARYING KOUNT FROM 1 BY 1
    UNTIL KOUNT > 12.

COUPON-PRINTING.
    MOVE KOUNT TO COUPON-NUMBER.
    MOVE MONTH (KOUNT) TO MONTH-NAME.
    MOVE AMOUNT TO EDITED-AMOUNT.
    WRITE PRINT-LINE FROM COUPON.
```

Incidentally, KOUNT is spelled with a K because COUNT, whose use was discussed in Chapter 8 on the UNSTRING verb, is a COBOL reserved word and

therefore cannot be used as a data-name. Also note that KOUNT, which is varied by PERFORM, is used in three ways:

1 To control the number of executions
2 As the coupon number
3 As the subscript to retrieve the corresponding month-name from MONTHS-TABLE

It now is appropriate to review the overall procedure by which the VARYING option is carried out, as already outlined in the flowchart in Figure 11-5. The procedure is as follows:

1 The identifier to be varied is set at its initial value, the value indicated by the clause:

$$\underline{\text{FROM}}\begin{Bmatrix}\text{identifier-2}\\\text{literal}\end{Bmatrix}$$

2 A test is made to determine if the condition specified by UNTIL is met. If it is met, PERFORM is skipped and control passes to the next statement. If the condition is not met, then the paragraph(s) specified is executed once.
3 The value of the varied identifier is incremented by the amount shown in the clause:

$$\underline{\text{BY}}\begin{Bmatrix}\text{identifier-3}\\\text{literal}\end{Bmatrix}$$

4 The procedure in steps 2 and 3 is repeated.

The condition need not refer to the value of the identifier-1, which is varied, even though the examples given illustrate only such cases. The condition can refer to other identifiers, but in all cases it must refer to identifiers that have their values altered by the paragraphs under PERFORM control. Otherwise, the loop will repeat indefinitely, as in the following example:

```
          MOVE 10 TO AMOUNT
          PERFORM ABC VARYING L FROM 1 BY 1
                 UNTIL AMOUNT > 20.
      ABC.
          WRITE REPORT LINE.
```

The problem with this segment is that, whereas the value of L is being incremented, the value of AMOUNT is being tested. Since AMOUNT never is altered by the ABC paragraph, there will be no end to the loop!

Now consider one more example that further illustrates use of the VARYING option and utilizes the STRING verb described in Chapter 8. Suppose that a header is to be centered with respect to column 40 of a printed page. The size of the header is variable, but it is always 20 or less characters long. The header is stored in the field called HEADER, and we wish to move it and print it from the output record called OUTPUT-RECORD.

Consider the following DATA and PROCEDURE DIVISION entries:

```
01  CHECK-FIELD          PIC X.
01  I                    PIC 99.
01  HEADER.
    02  INDIV-CHAR       PIC X OCCURS 20 TIMES.
        .
        .
        .
```

```
MOVE SPACE TO CHECK-FIELD
PERFORM DETERMINE-SIZE VARYING I FROM 20 BY -1
        UNTIL CHECK-FIELD NOT = SPACE
            OR I = ZERO.
ADD 1 TO I
COMPUTE I = 40 - (I / 2)
MOVE SPACES TO OUTPUT-RECORD
STRING HEADER DELIMITED BY SIZE
    INTO OUTPUT-RECORD
    WITH POINTER I.
WRITE OUTPUT-RECORD. . . .
    .
    .
    .
DETERMINE-SIZE.
    IF INDIV-CHAR (I) NOT = SPACE
        MOVE 'X' TO CHECK-FIELD.
```

The PERFORM DETERMINE-SIZE statement searches the HEADER field, character by character, from the right end of the field. When a nonblank character is encountered or the entire field has been searched, the search is terminated. A value of 1 is then added to I to restore it to the value that identifies the proper length. For example, if the data in HEADER consisted of ACME COMPANY, the Y character would cause X to be moved to CHECK-FIELD. Then, by nature of the PERFORM VARYING, I would be incremented by −1 and would become 11 before the UNTIL test was executed. Thus, the ADD 1 to I would restore I to the true length value of 12. Next, the procedure shows that we divide I by 2 and subtract this integer quotient from 40, which is the centering column. In this example, the data in HEADER is 12 characters long, thus, I would be I = 40 − 12/2 = 34. Then, use of the WITH POINTER I clause in the STRING verb would move the HEADER data into OUTPUT-RECORD, beginning with column 34.

Review

1 In general, use of the PERFORM . . . VARYING verb allows __________ execution of program modules.

repetitive

2 The object of PERFORM is not executed at all if the integer or the identifier used with PERFORM . . . VARYING has a __________ or __________ value.

zero; negative

3 When using PERFORM . . . VARYING, control of PERFORM is associated with systematically incrementing the value of a(n) __________.

identifier

4 The key programming word that indicates that an identifier is to be systematically incremented in value is the COBOL reserved word __________.

VARYING

5 The test made to determine if the condition specified for terminating PERFORM control has been met is indicated by the COBOL reserved word __________.

UNTIL

SAMPLE FORECASTING PROGRAM

The example we present in this section illustrates the application of PERFORM, OCCURS, and REDEFINES, and the use of tables and subscripts. The function of the program is to output a sales forecast. The input values are:

NEXT-MONTH A numeric value that designates the first month to be included in the forecast

HOW-MANY-MONTHS A 2-digit number that designates the number of months to be included in the forecast

BASE A dollar value used as the base for the forecast formula

COEFFICIENT A numeric coefficient used in the forecast formula

The forecasting formula used is:

$$F_i = B + cN$$

The forecast for month i (F_i) is equal to the base (B) plus a coefficient (c) times the number of months (N) from the starting point. If the first month is 2 (February), then the forecast for April will be:

$$F_{April} = B + c(2)$$

Thus, $N = 2$ in this case, since April is 2 months after February, which is the starting month.

If the following input were used, the resulting output would be as shown in Figure 11-6.

```
NEXT-MONTH        05
HOW-MANY-MONTHS   09
BASE              0010000000
COEFFICIENT       0000025^000
```

A sample forecast program listing is given in Figure 11-7. Notice that the setting up of the MONTH-TABLE in the WORKING-STORAGE SECTION is the same as presented in the earlier section of this chapter: "An Example of a Table of Constant Values."

In the PROCEDURE DIVISION, the paragraph called CALCULATION-ROUTINE is performed HOW-MANY-MONTHS times. In this case, the input field, HOW-MANY-MONTHS, contains the number of desired executions of the forecasting computation.

The MONTH-FROM-NOW field corresponds to the N in the forecasting formula $F_i = B + cN$. Finally, the subscript WHICH-ONE is used to reference the name of the month relating to each successive line of output. Since we have only 12 months, we may need to "wraparound" the MONTH-TABLE entries. For instance, if NEXT-MONTH = 5 and HOW-MANY-MONTHS = 9, the last month is not the thirteenth (4 + 9); rather, it is the first month of the next year. Thus, when the month subscript called WHICH-ONE exceeds 12, we subtract 12 from it to "bend" it down around the table.

```
                 PROJECTED SALES

MAY                         100250.00
JUNE                        100500.00
JULY                        100750.00
AUGUST                      101000.00
SEPTEMBER                   101250.00
OCTOBER                     101500.00
NOVEMBER                    101750.00
DECEMBER                    102000.00
JANUARY                     102250.00
```

FIGURE 11-6

ILLUSTRATIVE COMPUTER OUTPUT.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. FORECAST.
*
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT INPUT-DATA ASSIGN  TO READER.
    SELECT OUTPUT-FILE ASSIGN TO PRINTER.
*
DATA DIVISION.
FILE SECTION.
FD  INPUT-DATA LABEL RECORD OMITTED
               DATA RECORD IS INCARD.
01  INCARD.
    02 NEXT-MONTH        PICTURE 99.
    02 HOW-MANY-MONTHS PICTURE 99.
    02 BASE              PICTURE S9(8)V99.
    02 COEFFICIENT       PICTURE S9(8)V99.
    02 FILLER            PICTURE X(56).
*
FD  OUTPUT-FILE LABEL RECORD OMITTED
               DATA RECORD OUT-LINE.
01  OUT-LINE PICTURE X(133).
*
WORKING-STORAGE SECTION.
*
01  WHICH-ONE          PIC 99.
01  MONTHS-FROM-NOW    PIC 99.
01  SALES              PIC S9(9)V99.
01  DATA-END           PIC XXX.
*
01  HEADER.
    02 FILLER            PICTURE X(15) VALUE SPACES.
    02 FILLER            PICTURE X(15) VALUE 'PROJECTED SALES'.
*
01  PRINT-RECORD.
    02 FILLER            PICTURE X VALUE SPACE.
    02 MONTH-NAME        PICTURE X(12).
    02 FILLER            PICTURE X(5) VALUE SPACES.
    02 EDIT-SALES        PICTURE --------99.99.
01  MONTH-TABLE.
    02 JANUARY           PICTURE X(9) VALUE 'JANUARY  '.
    02 FEBRUARY          PICTURE X(9) VALUE 'FEBRUARY '.
    02 MARCH             PICTURE X(9) VALUE 'MARCH    '.
    02 APRIL             PICTURE X(9) VALUE 'APRIL    '.
    02 MAY               PICTURE X(9) VALUE 'MAY      '.
    02 JUNE              PICTURE X(9) VALUE 'JUNE     '.
    02 JULY              PICTURE X(9) VALUE 'JULY     '.
    02 AUGUST            PICTURE X(9) VALUE 'AUGUST   '.
    02 SEPTEMBER         PICTURE X(9) VALUE 'SEPTEMBER'.
    02 OCTOBER           PICTURE X(9) VALUE 'OCTOBER  '.
    02 NOVEMBER          PICTURE X(9) VALUE 'NOVEMBER '.
    02 DECEMBER          PICTURE X(9) VALUE 'DECEMBER '.
*
01  MONTHS REDEFINES MONTH-TABLE.
    02 MONTH             PICTURE X(9) OCCURS 12 TIMES.
/
PROCEDURE DIVISION.
MAIN-ROUTINE.
    OPEN INPUT INPUT-DATA
    OPEN OUTPUT OUTPUT-FILE
    MOVE 'NO' TO DATA-END
*
    READ INPUT-DATA RECORD
      AT END MOVE 'YES' TO DATA-END.
*
    IF DATA-END = 'NO'
       WRITE OUT-LINE FROM HEADER AFTER PAGE
```

FIGURE 11-7
SAMPLE FORECAST PROGRAM.

```
             MOVE SPACES TO OUT-LINE
             WRITE OUT-LINE AFTER ADVANCING 1 LINE
             MOVE ZERO TO MONTHS-FROM-NOW
             MOVE NEXT-MONTH TO WHICH-ONE
*
             PERFORM CALCULATION-ROUTINE
                     HOW-MANY-MONTHS TIMES
*
         ELSE
             MOVE 'NO DATA AVAILABLE' TO OUT-LINE
             WRITE OUT-LINE AFTER 1 LINE.
*
         CLOSE INPUT-DATA, OUTPUT-FILE.
         STOP RUN.
*
     CALCULATION-ROUTINE.
         ADD 1 TO MONTHS-FROM-NOW
         IF WHICH-ONE IS GREATER THAN 12
             SUBTRACT 12 FROM WHICH-ONE.
         COMPUTE SALES =
                 MONTHS-FROM-NOW * COEFFICIENT.
         ADD BASE TO SALES.
         MOVE MONTH (WHICH-ONE) TO MONTH-NAME.
         MOVE SALES TO EDIT-SALES.
         WRITE OUT-LINE FROM PRINT-RECORD
               AFTER ADVANCING 1 LINE.
         ADD 1 TO WHICH-ONE.
```

FIGURE 11-7 (*Continued*)
SAMPLE FORECAST PROGRAM.

SAMPLE PROGRAM WITH GRAPHIC OUTPUT

This section of the chapter presents a sample program that utilizes table-handling concepts to produce graphic printer output in the form of a bar chart. The output format is illustrated in Figure 11-8. For each common stock issue, we print a bar whose length corresponds to the percent yield of the stock.

The yield is computed as the percent ratio of the dividend-per-share to the price-per-share. It is assumed that no yield can exceed 50 percent, but if the computed yield does exceed this percentage, an error message is printed.

```
                                        PERCENT YIELD

                         Ø    5    1Ø   15   2Ø   25   3Ø   35   4Ø   45   5Ø
  STOCK NAME             I....I....I....I....I....I....I....I....I....I....I
                         I
FORD MOTOR CO.           IXXXXXXXXXX
                         I
GENERAL MOTORS CORP.     IXXXXXXXXXXXXX
                         I
CONTROL DATA CORP.       IXX
                         I
IBM CORP.                IXXXXXXX
                         I
SPERRY RAND CORP.        IXXX
                         I
HONEYWELL CORP.          IXXXX
                         I
DIGITAL EQUIPMENT CO     IXXXX
                         I
EXAMPLE ERROR-1          IINVALID INPUT DATA
                         I
EXAMPLE ERROR-2          IINVALID INPUT DATA
                         I
JACK-POT CORP.           IYIELD HIGHER THAN 5Ø%
```

FIGURE 11-8
SAMPLE GRAPHIC OUTPUT FOR THE STOCK-YIELD PROGRAM.

The input consists of records containing the stock name, the stock price, and the dividend. Figure 11-9 presents the program listing. The graphic output is prepared in the two paragraphs named FILL-BAR and MOVE-X-TO-BAR. Essentially, we move as many X's to the output line as the percent yield of each stock.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. GRAPH.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT STOCK-FILE  ASSIGN TO READER.
     SELECT REPORT-FILE ASSIGN TO PRINTER.
/
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  STOCK-FILE
           LABEL RECORDS OMITTED
           DATA RECORD IS STOCK-REC.
 01  STOCK-REC.
     02 STOCK-NAME          PIC X(20).
     02 STOCK-PRICE         PIC 9(3)V99.
     02 STOCK-DIVIDEND      PIC 9(2)V99.
     02 FILLER              PIC X(51).
*
 FD  REPORT-FILE
             LABEL RECORDS OMITTED
             DATA RECORD IS REPORT-REC.
 01   REPORT-REC            PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-SWITCH     PIC XXX VALUE 'NO'.
     88 END-OF-FILE VALUE 'YES'.
*
 01  PERCENT-YIELD          PIC 99.
*
 01  I                      PIC 99.
*
 01  GRAPH-LINE.
     02 FILLER              PIC X(5) VALUE SPACES.
     02 STOCK-NAME          PIC X(20).
     02 FILLER              PIC X(2) VALUE SPACES.
     02 FILLER              PIC X VALUE 'I'.
     02 BAR-CHART.
        03 BAR-CELL OCCURS 50 TIMES PIC X.
*
 01  HEADING-1.
     02 FILLER              PIC X(40) VALUE SPACES.
     02 FILLER              PIC X(13) VALUE 'PERCENT YIELD'.
  01  HEADING-2.
     02 FILLER              PIC X(27) VALUE SPACES.
     02 FILLER              PIC X(51) VALUE
       '0    5   10    15    20    25    30    35    40    45   50'.
*
 01  HEADING-3.
     02 FILLER              PIC X(10) VALUE SPACES.
     02 FILLER              PIC X(10) VALUE 'STOCK NAME'.
     02 FILLER              PIC X(7) VALUE SPACES.
     02 FILLER              PIC X VALUE 'I'.
     02 FILLER              PIC X(50) VALUE ALL '....I'.
```

FIGURE 11-9

LISTING OF THE PROGRAM FOR GRAPHIC OUTPUT.

```
      *
       01  EMPTY-LINE.
           02 FILLER                 PIC X(27) VALUE SPACES.
           02 FILLER                 PIC X VALUE 'I'.
      /
       PROCEDURE DIVISION.
      *
       PROGRAM-SUMMARY.
           OPEN INPUT STOCK-FILE
                OUTPUT REPORT-FILE.
      *
           PERFORM READ-STOCK-REC
      *
           WRITE REPORT-REC FROM HEADING-1 AFTER PAGE.
           WRITE REPORT-REC FROM HEADING-2 AFTER 2.
           WRITE REPORT-REC FROM HEADING-3 AFTER 2.
      *
           PERFORM PRINT-GRAPH
                   UNTIL END-OF-FILE
      *    CLOSE STOCK-FILE
                 REPORT-FILE
      *
           STOP RUN.
      *
       READ-STOCK-REC.
           READ STOCK-FILE RECORD
                AT END MOVE 'YES' TO END-OF-FILE-SWITCH.
      *
       PRINT-GRAPH.
           WRITE REPORT-REC FROM EMPTY-LINE
      *
           IF STOCK-PRICE NOT NUMERIC
              OR STOCK-DIVIDEND NOT NUMERIC
              OR STOCK-PRICE NOT > ZERO
             MOVE 'INVALID INPUT DATA' TO BAR-CHART
           ELSE
             COMPUTE PERCENT-YIELD ROUNDED
                    = STOCK-DIVIDEND * 100.0 / STOCK-PRICE
      *
             IF PERCENT-YIELD > 50
                MOVE 'YIELD HIGHER THAN 50%' TO BAR-CHART
             ELSE
                PERFORM FILL-BAR.
           MOVE STOCK-NAME OF STOCK-REC
                TO STOCK-NAME OF GRAPH-LINE
      *
           WRITE REPORT-REC FROM GRAPH-LINE.
      *
           WRITE REPORT-REC FROM EMPTY-LINE.
      *
           PERFORM READ-STOCK-REC.
      *
       FILL-BAR.
           MOVE SPACES TO BAR-CHART
           PERFORM MOVE-X-TO-BAR VARYING I FROM 1 BY 1
                                 UNTIL I > PERCENT-YIELD.
      *
       MOVE-X-TO-BAR.
           MOVE 'X' TO BAR-CELL (I).
```

FIGURE 11-9 *(Continued)*

LISTING OF THE PROGRAM FOR GRAPHIC OUTPUT.

■ INTERNAL SORTING

Internal sorting is concerned with sorting procedures (algorithms) used with tables of data held in central storage. Sorting an internal table is a different task from sorting data recorded on external files, such as disk or tape. There are many internal sort algorithms, and new ones continue to be developed for special cases. In this section we present two versions of the *interchange sort algorithm*.

There are several versions of the interchange sort approach. Fundamental to all versions is the interchange of two elements in the table. To interchange data in two fields we need a third field for temporary storage. For instance, if we want to interchange the data in T(X) and T(Y), we could write:

```
MOVE T(X) TO TEMP
MOVE T(Y) TO T(X)
MOVE TEMP TO T(Y).
```

Any interchange algorithm involves paired comparisons of table elements. The *adjacent comparison-interchange* approach involves comparison of adjacent table values and may require multiple "passes" through the table. In each pass we interchange adjacent elements that are out of order relative to each other. Specifically, in the first pass the first entry is compared with the second, the second with the third, and so on until the $(N - 1)$th record is compared with the Nth record in the table. The two records being compared each time are interchanged whenever the second record is smaller than the first record, assuming that the table is being sorted into ascending order. If no interchange was necessary during an entire pass, this means that the table is already in order. Otherwise, at least one more pass is required.

By the adjacent comparison approach, at the end of the first pass the largest record is driven to the bottom of the table. Thus in the second pass through the table $N - 1$ table entries are considered, in the third pass $N - 2$ are considered, and so forth. Thus the table is sorted from the last entry up, and every pass through the table involves a smaller table for comparisons and interchanges. Eventually only two elements remain to be considered—the first and the second.

Typically, however, the table will come to be sorted correctly before all possible passes through the table are made by the adjacent comparison approach. As indicated above, the fact that no interchange is necessary during a pass through the table serves to indicate that the table is already correctly sorted. In order to avoid unnecessary passes through the table, it is useful to incorporate a procedure by which such a situation can be detected. One way of doing this is to initialize a data-name with a value such as zero, and to change the value to 1 with the occurrence of any interchange. At the end of each pass the value of the data-name is tested. If the value is zero, the table is sorted. If the value is 1, at least one more pass is required.

The procedure associated with the adjacent comparison-interchange algorithm will now be illustrated by means of a COBOL program, which is presented in Figure 11-10.

The input consists of records, each of which consists of two fields, NAME-IN and FILLER. The NAME-IN will be used as the sort key while FILLER is assumed to contain other data. Data is read in by execution of the READ-DATA-IN SECTION, which stores the records in a table called ENTIRE-TABLE. Notice that N is a counter that represents the number of records read (maximum of 50).

The sorting operation takes place in the SORT-DATA SECTION. The PERFORM OUTER-LOOP is executed as long as TEST is not equal to SORTED and as long as I is not equal to N − 1. In essence, this instruction says to keep going through the table as long as it is not sorted after each pass through, but in any case not to go through the table more than N − 1 times. In the OUTER-LOOP paragraph, TEST is set to the value "SORTED" so that if no interchange takes place, that value will stay as such and will terminate execution of the OUTER-LOOP paragraph. Each time through, M is decreased by 1, since the table is effectively shortened as the largest value floats to the bottom. The PERFORM PAIRED-COMPARISONS instruction allows us to compare M − 1 pairs and interchange their values as needed. The PRINT-DATA SECTION simply lists the data on the printer for visual review.

Another version of the interchange sort algorithm is the *bubble sort-interchange*

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. INSORT.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT INPUT-FILE  ASSIGN TO READER.
     SELECT OUTPUT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
*
 FD  INPUT-FILE LABEL RECORD OMITTED
                DATA RECORD INPUT-RECORD.
 01  INPUT-RECORD.
     02 NAME-IN          PIC X(15).
     02 FILLER           PIC X(65).
 FD  OUTPUT-FILE LABEL RECORD OMITTED
                 DATA RECORD PRINT-RECORD.
 01  PRINT-RECORD        PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  ENTIRE-TABLE.
     02 TABLE-REC OCCURS 50 TIMES.
         03 NAME         PIC X(15).
         03 FILLER       PIC X(65).
 01  DATA-END            PIC XXX.
 01  TEST                PIC X(8).
 01  N                   PIC 99.
 01  I                   PIC 99.
 01  J                   PIC 99.
 01  K                   PIC 99.
 01  M                   PIC 99.
 01  TEMP-STORE          PIC X(80).
/
 PROCEDURE DIVISION.
*
 MAIN-ROUTINE.
     OPEN INPUT INPUT-FILE
          OUTPUT OUTPUT-FILE
*
     PERFORM READ-DATA-IN
*
     PERFORM SORT-DATA.
*
     PERFORM PRINT-DATA.
*
     CLOSE INPUT-FILE, OUTPUT-FILE
     STOP RUN.
 READ-DATA-IN SECTION.
*
 SET-UP-TO-READ.
     MOVE 'NO' TO DATA-END.
     MOVE ZERO TO N.
     PERFORM READ-DATA
*
     PERFORM STORE-READ
         UNTIL DATA-END = 'YES'
            OR N = 50.
     GO TO EXIT-READ.
*
 READ-DATA.
     READ INPUT-FILE AT END MOVE 'YES' TO DATA-END.
*
 STORE-READ.
     ADD 1 TO N
     MOVE INPUT-RECORD TO TABLE-REC (N)
     PERFORM READ-DATA.
```

FIGURE 11-10

COBOL INTERNAL SORT PROGRAM UTILIZING THE ADJACENT COMPARISON-INTERCHANGE ALGORITHM.

```
*
 EXIT-READ.
     EXIT.
*
 SORT-DATA SECTION.
*
 SET-UP-TO-SORT.
     MOVE 'UNSORTED' TO TEST
     MOVE N TO M
*
     PERFORM OUTER-LOOP VARYING I FROM 1 BY 1
                              UNTIL TEST = 'SORTED'
                                      OR I = N - 1.
     GO TO SORT-ENDED.
*
 OUTER-LOOP.
     MOVE 'SORTED' TO TEST
     COMPUTE M = M - 1
     PERFORM PAIRED-COMPARISONS VARYING J FROM 1 BY 1
                              UNTIL J > M.
*
 PAIRED-COMPARISONS.
     COMPUTE K = + J
*
     IF NAME (J) IS GREATER THAN NAME (K)
         MOVE TABLE-REC (J) TO TEMP-STORE
         MOVE TABLE-REC (K) TO TABLE-REC (J)
         MOVE TEMP-STORE TO TABLE-REC (K)
         MOVE 'UNSORTED' TO TEST
     ELSE
         NEXT SENTENCE.
 SORT-ENDED.
     EXIT.
*
 PRINT-DATA SECTION.
*
 PRINT-ROUTINE.
     PERFORM PRINTOUT VARYING I FROM 1 BY 1
                              UNTIL I IS GREATER THAN N.
     GO TO PRINT-END.
*
 PRINTOUT.
     MOVE TABLE-REC (I) TO PRINT-RECORD
     WRITE PRINT-RECORD AFTER ADVANCING 1 LINE.
*
 PRINT-END.
     EXIT.
```

FIGURE 11-10 *(Continued)*
COBOL INTERNAL SORT PROGRAM UTILIZING THE ADJACENT COMPARISON-INTERCHANGE ALGORITHM.

algorithm. Use of this method results in the first part of the table being sorted first. By the bubble sort-interchange method the first step is that the second record is compared with the first, and if necessary, they are interchanged. Then the third record is compared with the second. If these two records are interchanged, the (new) second record is compared with the first, interchanging if necessary. Next, the fourth and third record are compared for possible interchange. Again, if an interchange occurs, we go "upward" and compare the third with the second record and then possibly the second with the first record. Thus, at each stage a record rises like a bubble to find its proper place, and hence the name "bubble sort" which is applied to this algorithm.

As an example of using the bubble sort-interchange algorithm, consider the following set of six records with sort keys as indicated:

5
8
10
11
9
15

Comparison of the second and first records results in no interchange. Similarly, comparison of the third and second and of the fourth and third records results in no interchange. However, when the fifth and fourth records are compared (9 and 11) an interchange results. Following this, comparison of the fourth and third records (9 and 10) results in another interchange, but then the comparison of the third and second records (9 and 8) results in no further interchange. Finally, the sixth and fifth records are compared (15 and 11). Since this is the last pair of records and no interchange is required, the sort routine is completed.

Review

1 By "internal sorting" we mean the sorting of a table held in the ________ of the computer.

internal (or central) storage

2 Several versions of the interchange sort algorithm exist. The feature which is common to all these versions is that in each approach ____________ (number) records are compared each time for possible interchange.

2

3 By the adjacent comparison-interchange algorithm all adjacent records are compared and interchanged, when necessary, during each pass through the table. The fact that the table is correctly sorted is indicated when ________ (number) interchanges are required during a particular pass through the table.

0

4 Another version of the interchange sort algorithm is the bubble sort-interchange. By this approach, each record is properly placed before subsequent records are considered, resulting in the [first / last] part of the table being sorted first and thereby requiring ____________ (number) passes for a table of *N* records.

first; 1

EXERCISES

11.1 Write DATA DIVISION entries to set up a table that is to contain annual dollar sales for the years 1970–1982. No value will exceed $100,000,000.00.

11.2 Write DATA DIVISION entries to set up a table to contain dollar and unit sales for the years 1975–1987. We want to be able to reference the dollar sales or the unit sales individually for each year, as well as to reference as a group the dollar sales and unit sales pertaining to a given year. The general format of the table is as follows:

YEAR	DOLLAR SALES	UNIT SALES
1975		
1976		
.		
.		
.		
1987		

11.3 Use DATA DIVISION entries to form a table containing the names of the days of the week so that the names are referenced by a subscript; thus, Monday would have a subscript 1 and Sunday would have a subscript 7.

11.4 Assume that TAX-TABLE contains 30 values (V999). Write the PROCEDURE and DATA DIVISION statements required to print the contents of the table in the following formats:

- **a** Print the 30 values in one column of 30 lines.
- **b** Print the 30 values at the rate of seven per line for as many lines as are needed.

11.5 For the following table, write the necessary program instructions to find the smallest value and to place it in SMALLEST. Disregard the possibility of ties.

```
02 TABLE OCCURS 50 TIMES     PICTURE X(12).
```

11.6 A sales file contains the following types of data:

COLUMNS	FIELD DESCRIPTION
1	Quarter during which sales occurred; PIC 9, values are in range of 1-4.
2–6	Sales amount; PIC 999V96.

There may be many records referring to the same quarter. Write a program to read such a sales file and produce the following type of report:

QUARTER	TOTAL SALES	% OF YEAR TOTAL
1	100.00	20%
2	50.00	10%
3	300.00	60%
4	50.00	10%
YEAR TOTAL	$500.00	100%

Each value in the TOTAL SALES column may represent the sum of several sales values.

11.7 A customer file contains customer names, among other data. We are interested in conducting a frequency analysis of the first letter of customer names. In other words, a table such as the following is to be produced:

BEGINNING LETTER	NUMBER OF NAMES
A	10
B	15
.	.
.	.
.	.
Z	1

Input records have the following format:

COLUMNS	FIELD DESCRIPTION
1–12	Customer name
13–80	Other data

Write a program to do the above task.

11.8 Modify the sample program presented in Figure 11-10 to employ the bubble sort method instead of the adjacent comparison method for sorting records.

12 ADDITIONAL TABLE-HANDLING FEATURES

TWO-DIMENSIONAL AND HIGHER-DIMENSIONAL TABLES

The concept of "dimensions" refers to categories by which data is organized within tables. As an example, suppose that we have student enrollment data organized as follows:

By college
  By department within colleges
    By field of specialization within departments
      By year of studies (freshman through senior) within field of specialization

Data so organized in a table would be in four dimensions: by college, department, field of specialization, and year of studies. Thus, when data are organized in nested categories we refer to such categories as dimensions.

In the 1974 standard version of COBOL, the maximum number of dimensions permitted is three. In the revised version, the programmer can use up to seven dimensions. In practice, the need for more than three dimensions is rare. Therefore either version of the language is powerful enough for handling most higher-dimensioned tables.

Two-dimensional tables require two subscripts to locate an individual entry, or field. As an example of a two-dimensional table, assume that a particular state has three state universities and we desire to set up a table that will contain the enrollment figures for each university according to class standing: freshman, sophomore, junior, senior, and graduate. Figure 12-1 portrays the required table.

<table>
<tr><td colspan="4">ENROLLMENT BY UNIVERSITY</td></tr>
<tr><td>YEAR</td><td>1</td><td>2</td><td>3</td></tr>
<tr><td>FR</td><td></td><td></td><td></td></tr>
<tr><td>SO</td><td></td><td></td><td></td></tr>
<tr><td>JR</td><td></td><td></td><td></td></tr>
<tr><td>SR</td><td></td><td></td><td></td></tr>
<tr><td>GR</td><td></td><td></td><td></td></tr>
</table>

FIGURE 12-1

REQUIRED FORMAT OF THE ENROLLMENT TABLE.

In order to set up the required storage locations, the following DATA DIVISION entries can be written:

```
01  ENROLLMENT-TABLE.
    02  UNIVERSITY-ENROLLMENT   OCCURS  3  TIMES.
        03  YEAR-ENROLLMENT     OCCURS  5  TIMES.
            04  ENROLLMENT      PIC  9(5).
```

Two dimensions are specified in the statements above. YEAR-ENROLLMENT, which is at the 03 level and includes an OCCURS clause, is under UNIVERSITY-ENROLLMENT, which is at the 02 level and also includes an OCCURS clause. Thus, a data definition statement that involves an OCCURS which is subordinate to another OCCURS constitutes a (nested) higher dimension.

Notice the naming conventions illustrated in the above example. The entire collection of data is called ENROLLMENT-TABLE, which seems an appropriate data-name. The first dimension is named UNIVERSITY-ENROLLMENT, a name corresponding to the type of data, while the second dimension is given the data-name YEAR-ENROLLMENT. The 04 ENROLLMENT data-name is at a lower level (04), but since it is the only item at that level, it is really the same as YEAR-ENROLLMENT. However, we use the additional data-name for documentation purposes. The elementary data items refer to enrollment, and so we wish to use such a name for the data item that has the PIC clause. The 02-level and 03-level items are simply ways of organizing the data.

It is instructive to consider the storage layout for the above table. Figure 12-2 illustrates such a table containing hypothetical enrollment data. For instance, we can observe that the junior-year enrollment for the first of the three universities is 1000 students. As the figure illustrates, enrollment data for each of the five enrollment years is stored separately for each of the universities. Thus it is critical as to which OCCURS clause is written first. In our example UNIVERSITY-ENROLLMENT is at the 02 level followed by YEAR-ENROLLMENT at the 03 level. Accordingly, the data for each university is physically grouped together.

<table>
<tr><td colspan="27">ENROLLMENT TABLE</td></tr>
<tr><td colspan="26">UNIVERSITY 1</td><td>UNIVERSITY 2</td></tr>
<tr><td colspan="5">YEAR 1</td><td colspan="5">YEAR 2</td><td colspan="5">YEAR 3</td><td colspan="5">YEAR 4</td><td colspan="6">YEAR 5</td><td>. . .</td></tr>
<tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>5</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>2</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>3</td><td>0</td><td>0</td><td>. . .</td></tr>
</table>

FIGURE 12-2

STORAGE LAYOUT OF THE TWO-DIMENSIONAL TABLE

To illustrate the use of the table, consider the following examples.

UNIVERSITY-ENROLLMENT (1) refers to all five fields associated with the first university. Keep in mind that this is a reference to a group item, and in COBOL all group items are considered alphanumeric for MOVE purposes. There would be nothing wrong saying MOVE SPACES TO UNIVERSITY-ENROLLMENT (1) since the subscripted data-name refers to a 25-byte (5×5) alphanumeric field, not to a numeric field. This is consistent with the fact that each of the five individual enrollment figures for each university is a number. However, the five numbers together do not have any numeric (quantitative) meaning.

ENROLLMENT (1, 3) refers to the junior-year (3) enrollment at the first (1) university. ENROLLMENT must always be used with double subscripts. The first subscript refers to the superior dimension defined by the first OCCURS clause, which in this case is UNIVERSITY-ENROLLMENT. The second subscript refers to the subordinate dimension defined by the second OCCURS clause, YEAR-ENROLLMENT.

ENROLLMENT (UNIVERSITY, YEAR) refers to the enrollment of the university specified by the value of the UNIVERSITY field and the year defined by the value of YEAR. In order to be correct, UNIVERSITY must have a value in the range 1-3 and YEAR a value in the range 1-5. These ranges correspond to the OCCURS clauses written in the definition of the table.

YEAR-ENROLLMENT (1, 3) and YEAR-ENROLLMENT (UNIVERSITY, YEAR) are equivalent to the two examples above, respectively. However, we recommend using YEAR-ENROLLMENT for documentation purposes and using ENROLLMENT as the more appropriate data-name for the elementary item.

Three-dimensional tables involve the use of three subscripts. We can illustrate the statements required to set up a three-dimensional table by adding a further breakdown by male and female to our two-dimensional example, as follows:

```
01  ENROLLMENT-TABLE.
    02  UNIVERSITY-ENROLLMENT   OCCURS  3  TIMES.
        03  YEAR-ENROLLMENT     OCCURS  5  TIMES.
            04  SEX-ENROLLMENT  OCCURS  2  TIMES.
                05  ENROLLMENT  PIC  9(5).
```

Thus, ENROLLMENT (2, 3, 1) refers to the enrollment of males (1) in the junior year (3) in the second university (2).

The following rules serve to summarize the requirements associated with the use of subscripted tables.

1 The OCCURS clause cannot apply to the 01 level. In other words, there cannot be a table of "records." However, this is a language rule and in no way prevents us from assigning a "record" to the 02 level and defining an 01 level name above it, as shown in the following example:

```
01  STATE-INCOME-TABLE.
    02  AVERAGE-INCOME      OCCURS 50 TIMES      PICTURE 9(6)V99.
```

2 The OCCURS clause cannot be used with level 77 items.

3 Subscripted tables may have one, two, or three dimensions in the 1974 standard, or up to seven dimensions in the revised version of COBOL.

4 The PICTURE clause applies to the elementary items only. Notice, for instance, the example of the three-dimensional table given previously.

5 Only one PICTURE description can be given for all like entries in a table, but there may be several entries that are not alike. The latter is exemplified by the example given earlier and reproduced here:

```
01 STATE-INCOME-TABLE.
   02 NAME-INCOME       OCCURS 50 TIMES.
      03 NAME           PICTURE X(12).
      03 INCOME         PICTURE 9(6)V99.
```

The same PICTURE clause applies to all 50 Name fields above. Thus, OCCURS is used for homogeneous sets of data.

6 The subscripts may be integer constants, or they may be integer variables. Their values must be positive; they must not be zero or negative.

7 The subscript or subscripts are enclosed in one set of parentheses and are separated from the table-name by a space. Multiple subscripts are separated from each other by *commas* and *spaces*. Examples are:

```
A-TABLE (1)
A-TABLE (IDEN)
B-TABLE (3, COUNT)
C-TABLE (GRADE, CLASS, YEAR)
C-TABLE (GRADE, 3, YEAR)
D-TABLE (GRADE, CLASS, YEAR, STATE)
```

Review

1 In the two-dimensional ENROLLMENT-TABLE example, suppose we want the university to be at the lowest hierarchical level, instead of the year. Write the appropriate statements below, using a PICTURE of 9(5) for the elementary field.

```
01 ENROLLMENT-TABLE.
   02 ________________________________
      03 _____________________________
         04 __________________________
```

YEAR-ENROLLMENT OCCURS 5 TIMES.
UNIVERSITY-ENROLLMENT OCCURS 3 TIMES.
ENROLLMENT PICTURE 9(5)

2 For the example in the preceding question, the table location of juniors in the first university is referenced by the subscripted variable ______________.

UNIVERSITY-NUMBER (3, 1)

3 The PICTURE clause is used only at the [highest / lowest] hierarchical level of a table.

lowest

4 The integer subscripts used in conjunction with subscripted variables [may / may not] be constant and [may / may not] be variables.

may; may

5 A subscript used in conjunction with subscripted variables [may / may not] have a negative value.

may not

THE PERFORM VERB AND TABLE HANDLING

In the preceding chapter, we described the use of the PERFORM verb for table handling. We will continue the description by presenting a format of the PERFORM verb that is particularly useful for processing two-dimensional and higher-

```
PERFORM [procedure-name-1 [{THROUGH} procedure-name-2]]
                           {THRU   }

[with TEST {BEFORE}]
           {AFTER }

VARYING identifier-1 FROM {identifier-2}
                          {literal-1   }

   BY {identifier-3} UNTIL condition-1
      {literal-2   }

[ AFTER identifier-4 FROM {identifier-5}
                          {literal-3   }

   BY {identifier-6} UNTIL condition-2 ] . . .
      {literal-4   }

   [imperative-statement-1 END-PERFORM]
```

FIGURE 12-3
EXTENDED FORMAT OF THE PERFORM . . . VARYING STATEMENT.

dimensional tables. The extended format of PERFORM is presented in Figure 12-3. Consider the following example, which illustrates the use of two subscripts.

Assume that we have 20 source records, defined as:

```
FD  SOURCE-FILE LABEL RECORD OMITTED DATA RECORD IS SOURCE-REC.
01  SALES VALUE.
    02  SALES-VALUE     PICTURE 9(5)V99.
    02  FILLER          PICTURE X(73).
```

The data are sales values in each of the five sales territories for each of the last four quarters. The records are ordered so that the first record contains the sales value for territory 1, quarter 1; the second record the value for territory 1, quarter 2, and so on. Therefore, the twentieth record contains the value for territory 5, quarter 4. We want to read the records and store the values in a two-dimensional table as presented in Figure 12-4. The values in records 1–4 are to be stored in row 1, columns 1–4 (according to record sequence). Record values 5–8 are to be stored in row 2, columns 1–4, and so on. Let us first define the table:

```
01  SALES-DATA-TABLE.
    02  SALES-TERRITORY    OCCURS 5 TIMES.
        03  QUARTER-SALES  OCCURS 4 TIMES
                           PICTURE 9(5)V99.
```

	SALES TERRITORY				
QUARTER	1	2	3	4	5
1					
2					
3					
4					

FIGURE 12-4
REQUIRED FORMAT OF THE SALES TABLE.

SALES TERRITORY	QUARTER
1	1
1	2
1	3
1	4
2	1
2	2
2	3
2	4
.	.
.	.
.	.
5	1
5	2
5	3
5	4

FIGURE 12-5
SEQUENCE OF VALUES IN SALES-DATA-TABLE.

Since the table is two-dimensional, we will use two subscripts. Let these be defined as:

```
01 TERRITORY     PICTURE 9.
01 QUARTER       PICTURE 9.
```

Now we can write PROCEDURE DIVISION statements to accomplish the desired input-storing task, as follows:

```
PERFORM READER
    VARYING TERRITORY FROM 1 BY 1
        UNTIL TERRITORY > 5
    AFTER QUARTER FROM 1 BY 1
        UNTIL QUARTER > 4.
            .
            .
            .
READER.
    READ SOURCE-FILE
        AT END MOVE 'YES' TO END-OF-FILE-SWITCH.
    IF NOT END-OF-FILE
        MOVE SALES-VALUE
            TO QUARTER-SALES (TERRITORY, QUARTER).
```

Notice that TERRITORY is varied *after* varying QUARTER. Therefore the sequence of values contained in these two fields is as presented in Figure 12-5.

The paragraph called READER will be executed 20 times, as QUARTER and TERRITORY are varied through their specified ranges. Each time the READER paragraph is executed, the pair of values of the subscripts is unique, so that each value is stored in a new QUARTER-SALES cell of the table.

With respect to the general format of the PERFORM statement in Figure 12-3, when procedure-name-1 is specified in the revised version of COBOL, the imperative-statement-1 and the END-PERFORM are omitted. Conversely, when procedure-name-1 is omitted then imperative-statement-1 must be present, and the AFTER phrase must not be used. As an example of the last case we could have:

```
PERFORM VARYING I FROM 1 BY 1
        UNTIL I > 10
    ADD SALES (I) TO TOTAL-SALES
    IF SALES (I) > 0
        ADD 1 TO NON-ZERO-COUNT
    ELSE
        ADD 1 TO ZERO-COUNT
    END-IF
END-PERFORM.
```

In the above example, we form the sum of the first 10 values in SALES and we count the number of SALES values as nonzero and zero, respectively. In this case "imperative-statement-1" consists of the ADD and the IF statements, illustrating the fact that "imperative-statement" can consist of multiple statements, and can include conditional statements as long as they are delimited by an explicit scope terminator (END-IF in our example).

Figures 12-6 and 12-7 show the effect of using WITH TEST BEFORE and WITH TEST AFTER, respectively, in flowchart form. As can be seen, the

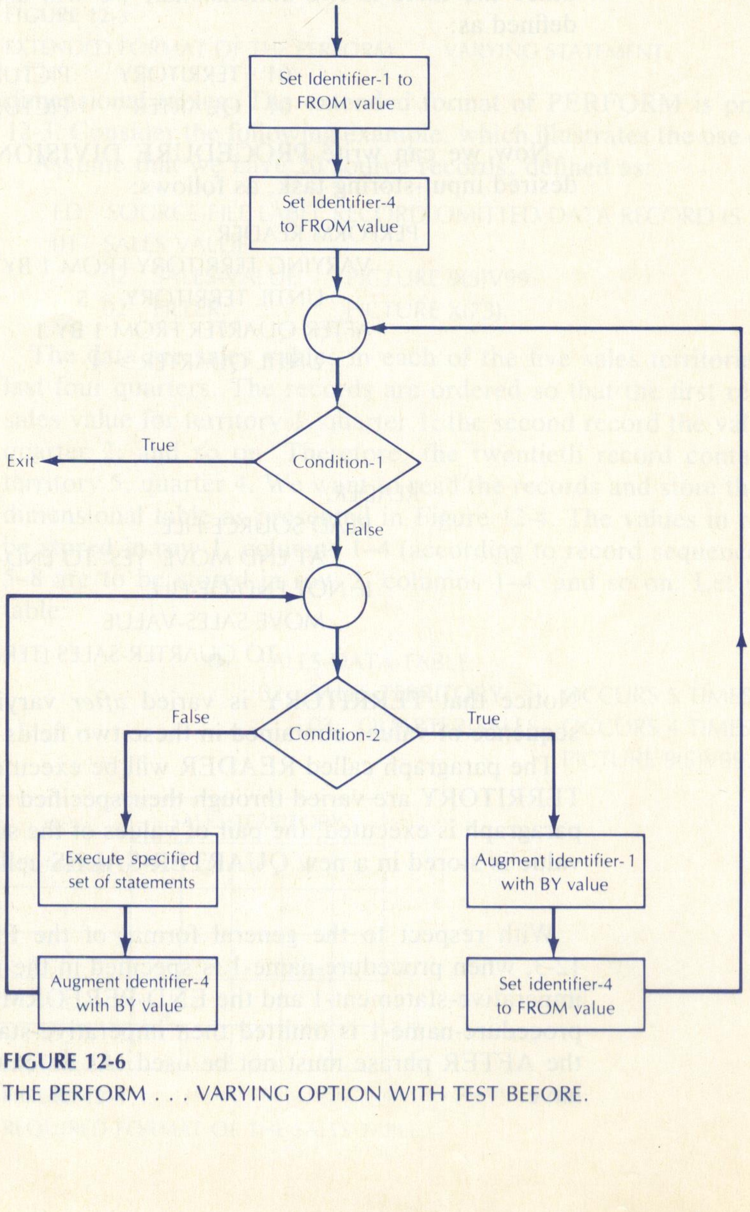


FIGURE 12-6

THE PERFORM . . . VARYING OPTION WITH TEST BEFORE.

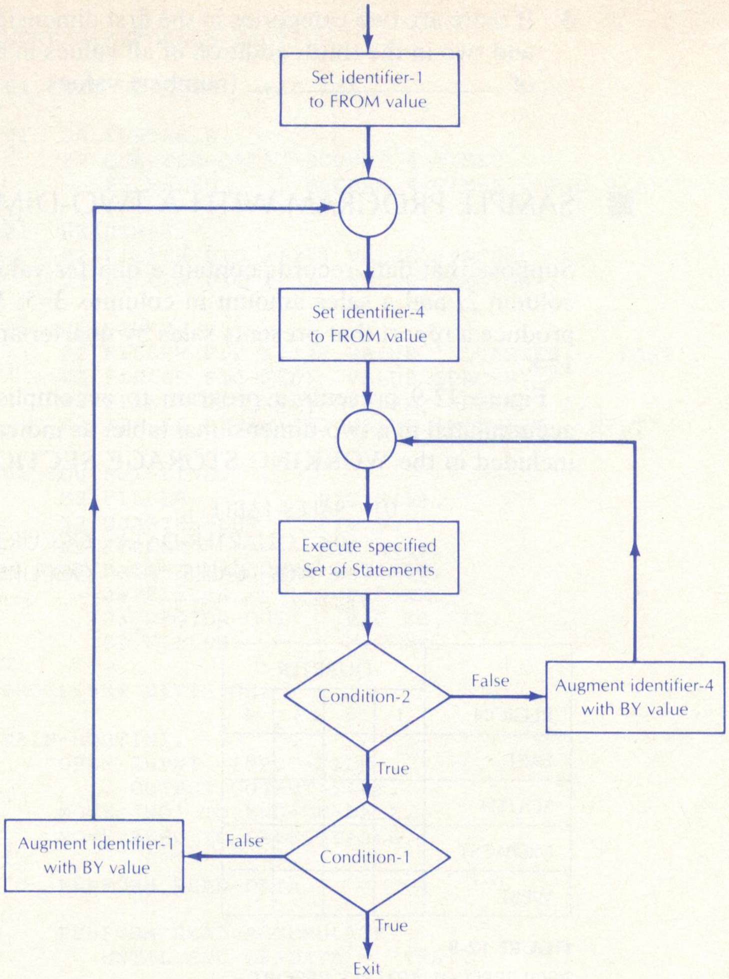


FIGURE 12-7
THE PERFORM . . . VARYING OPTION WITH TEST AFTER.

difference is that with TEST BEFORE the specified statements may not be executed even once if the conditions are true. On the other hand, the TEST AFTER option executes the specified statements at least once before testing whether or not execution should be repeated.

Omission of both the BEFORE and AFTER options is understood to imply the BEFORE option.

Review

1 The enhanced format of the PERFORM verb described in this section extends the efficiency of COBOL by allowing the programmer systematically to vary up to ________________ (number) identifiers in a nested fashion in the 1974 standard, and up to ________________ (number) in the revised version of COBOL.

3; 7

2 Each identifier may represent a subscript. For a two-dimensional table, for example, ________________ (number) subscripts are involved.

2

3 If there are two categories in the first dimension of a table, three in the second, and two in the third, addition of all values in the table involves the summation of ____________ (number) values.

$2 \times 3 \times 2 = 12$

SAMPLE PROGRAM WITH A TWO-DIMENSIONAL TABLE

Suppose that data records contain a quarter value in column 1, a region value in column 2, and a sales amount in columns 3–5. We want to read such a file and produce a report that presents sales by quarter and region, as illustrated in Figure 12-8.

Figure 12-9 presents a program to accomplish the task. The sales data are accumulated in a two-dimensional table, as indicated by the following description included in the WORKING-STORAGE SECTION of the DATA DIVISION:

```
01 SALES-TABLE.
    02 QUARTER-DATA  OCCURS 4 TIMES
       03 SALES      OCCURS 4 TIMES   PICTURE 9(5).
```

	QUARTER			
REGION	1	2	3	4
EAST				
SOUTH				
MIDWEST				
WEST				

FIGURE 12-8
REQUIRED QUARTERLY REPORT.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. TABLES.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT INPUT-FILE ASSIGN  TO READER.
     SELECT OUTPUT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
*
 FD  INPUT-FILE LABEL RECORD OMITTED
                     DATA RECORD IS INPUT-RECORD.
 01  INPUT-RECORD.
     02 QUARTER          PIC 9.
     02 REGION           PIC 9.
     02 AMOUNT           PIC 999.
     02 FILLER           PIC X(75).
*
 FD  OUTPUT-FILE LABEL RECORD IS OMITTED
                     DATA RECORD IS OUTPUT-RECORD.

 01  OUTPUT-RECORD PIC X(132).
```

FIGURE 12-9
PROGRAM FOR A QUARTERLY REPORT.

```
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-DATA   PIC XXX.
*
 01  SALES-TABLE.
     02 QUARTER-DATA  OCCURS 4 TIMES.
     03 SALES         OCCURS 4 TIMES    PIC 9(5).
*
 01  HEADER-1.
     02 FILLER PIC X(27) VALUE SPACES.
     02 FILLER PIC X(6)  VALUE 'REGION'.
*
 01  HEADER-2.
     02 FILLER PIC X(17) VALUE ' QUARTER    EAST'.
     02 FILLER PIC X(6)  VALUE SPACES.
     02 FILLER PIC X(9)  VALUE 'SOUTH    '.
     02 FILLER PIC X(15) VALUE 'MIDWEST   WEST'.
*
 01  OUTPUT-LINE.
     02 FILLER          PIC X(5).
     02 QUARTER-OUT     PIC 9.
     02 FILLER          PIC X(4).
     02 DATA-OUT OCCURS 4 TIMES.
        03 FILLER          PIC XX.
        03 REGION-OUT      PIC ZZ,ZZZ.
        03 FILLER          PIC XX.
/
 PROCEDURE DIVISION.
*
 MAIN-ROUTINE.
     OPEN INPUT  INPUT-FILE
          OUTPUT OUTPUT-FILE.
     MOVE 'NO' TO END-OF-DATA.
     MOVE ZERO TO SALES-TABLE.
*
     PERFORM READ-DATA.
*
     PERFORM READ-ACCUMULATE
        UNTIL END-OF-DATA = 'YES'.
*
     PERFORM HEADERS
*
     PERFORM PRINT-TABLE VARYING QUARTER FROM 1 BY 1
                        UNTIL QUARTER IS GREATER THAN 4.
*
     CLOSE INPUT-FILE, OUTPUT-FILE.
     STOP RUN.
*
 READ-DATA.
     READ INPUT-FILE AT END MOVE 'YES' TO END-OF-DATA.
*
 READ-ACCUMULATE.
     ADD AMOUNT TO SALES (QUARTER, REGION)
     PERFORM READ-DATA.
*
 HEADERS.
     WRITE OUTPUT-RECORD FROM HEADER-1 AFTER ADVANCING PAGE
     WRITE OUTPUT-RECORD FROM HEADER-2
        AFTER ADVANCING 3 LINES.
*
 PRINT-TABLE.
     MOVE SPACES TO OUTPUT-LINE.
     PERFORM MOVE-DATA VARYING REGION FROM 1 BY 1
                      UNTIL REGION IS GREATER THAN 4.
*
     MOVE QUARTER TO QUARTER-OUT.
     WRITE OUTPUT-RECORD FROM OUTPUT-LINE
        AFTER ADVANCING 2 LINES.
*
 MOVE-DATA.
     MOVE SALES (QUARTER, REGION) TO REGION-OUT (REGION).
```

FIGURE 12-9 (*Continued*)
PROGRAM FOR A QUARTERLY REPORT.

SAMPLE PROGRAM WITH A THREE-DIMENSIONAL TABLE

Figure 12-10 presents the listing of a program that illustrates the processing of a three-dimensional table. The program task is to read source data representing enrollment statistics and accumulate the enrollment in a three-dimensional ENROLLMENT-TABLE. As can be seen in the definition of the table, the data is organized by semester within department, within college.

The LOAD-DATA routine contains the characteristic instruction ADD NUMBER-ENROLLED TO ENROLLMENT (COLLEGE-IN, DEPT-IN, SEMESTER-IN).

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. 3-D-TABLE.
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT SOURCE-FILE   ASSIGN TO READER.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  SOURCE-FILE
     LABEL RECORDS OMITTED
     DATA RECORD IS SOURCE-RECORD.
 01  SOURCE-RECORD.
     02 COLLEGE-IN          PIC 9.
     02 DEPT-IN             PIC 9.
     02 SEMESTER-IN         PIC 9.
     02 NUMBER-ENROLLED     PIC 99.
     02 FILLER              PIC X(75).
*
 WORKING-STORAGE SECTION.
*
 01  ENROLLMENT-TABLE.
*
     02 COLLEGE-DATA OCCURS 3 TIMES.
*
         03 DEPARTMENT-DATA OCCURS 4 TIMES.
*
             04 SEMESTER-DATA OCCURS 2 TIMES.
*
                 05 ENROLLMENT                        PIC 9(4).
*
 01  END-OF-FILE        PIC XXX VALUE 'NO'.
 01  SUBSCRIPTS.
     02 COLLEGE         PIC 9.
     02 DEPT            PIC 9.
     02 SEMESTER        PIC 9.
*
 01  TOTALS-TABLES.
     02 COLLEGE-TOTALS-TABLE.
         03 COLLEGE-TOTALS-DATA OCCURS 3 TIMES.
             04 BLANK1                                    PIC XXX.
             04 COLLEGE-TOTALS                            PIC 9(4).
     02 DEPT-TOTALS-TABLE.
         03 DEPT-TOTALS-DATA  OCCURS 4 TIMES.
             04 BLANK2                                    PIC XXX.
             04 DEPT-TOTALS                               PIC 9(4).
     02 SEMESTER-TOTALS-TABLE.
         03 SEMESTER-TOTALS-DATA OCCURS 2 TIMES.
             04 BLANK3                                    PIC XXX.
             04 SEMESTER-TOTALS                           PIC 9(4).
     02 ENROLLMENT-TOTAL                                PIC 9(4).
```

FIGURE 12-10

SAMPLE PROGRAM WITH A THREE-DIMENSIONAL TABLE.

```
/
 PROCEDURE DIVISION.
*
 MAIN-ROUTINE.
     OPEN INPUT SOURCE-FILE
     PERFORM ZERO-ENROLLMENT-TABLES
     PERFORM READ-DATA
     PERFORM LOAD-DATA UNTIL END-OF-FILE = 'YES'
*
     PERFORM COLLEGE-SUM
     PERFORM DEPARTMENT-SUM
     PERFORM SEMESTER-SUM
     PERFORM TOTAL-ENROLLMENT-SUM
*
     CLOSE SOURCE-FILE
     STOP RUN.
*
 READ-DATA.
     READ SOURCE-FILE RECORD
          AT END MOVE 'YES' TO END-OF-FILE.
*
 LOAD-DATA.
     ADD NUMBER-ENROLLED TO
         ENROLLMENT (COLLEGE-IN, DEPT-IN, SEMESTER-IN).
     PERFORM READ-DATA.
*
 ZERO-ENROLLMENT-TABLES.
     MOVE ZERO TO ENROLLMENT-TOTAL
                  ENROLLMENT-TABLE
                  COLLEGE-TOTALS-TABLE
                  DEPT-TOTALS-TABLE
                  SEMESTER-TOTALS-TABLE.
*
 COLLEGE-SUM.
     PERFORM ACCUMULATE-COLLEGE-TOTALS
         VARYING COLLEGE FROM 1 BY 1
             UNTIL COLLEGE > 3
*
         AFTER DEPT FROM 1 BY 1
             UNTIL DEPT > 4
*
         AFTER SEMESTER FROM 1 BY 1
             UNTIL SEMESTER > 2.
     DISPLAY 'COLLEGE TOTALS    ', COLLEGE-TOTALS-TABLE.
*
 ACCUMULATE-COLLEGE-TOTALS.
     ADD ENROLLMENT (COLLEGE, DEPT, SEMESTER)
         TO COLLEGE-TOTALS (COLLEGE).
     MOVE SPACES TO BLANK1 (COLLEGE).
 DEPARTMENT-SUM.
     PERFORM ACCUMULATE-DEPT-TOTALS
         VARYING DEPT FROM 1 BY 1
             UNTIL DEPT > 4
*
         AFTER COLLEGE FROM 1 BY 1
             UNTIL COLLEGE > 3

*
         AFTER SEMESTER FROM 1 BY 1
             UNTIL SEMESTER > 2.
*
     DISPLAY 'DEPARTMENT TOTALS', DEPT-TOTALS-TABLE.
*
 ACCUMULATE-DEPT-TOTALS.
     ADD ENROLLMENT (COLLEGE, DEPT, SEMESTER)
         TO DEPT-TOTALS (DEPT).
     MOVE SPACES TO BLANK2 (DEPT).
*
 SEMESTER-SUM.
     PERFORM ACCUMULATE-SEMESTER-TOTALS
         VARYING SEMESTER FROM 1 BY 1
             UNTIL SEMESTER > 2
```

FIGURE 12-10 *(Continued)*
SAMPLE PROGRAM WITH A THREE-DIMENSIONAL TABLE.

```
      *
                        AFTER COLLEGE FROM 1 BY 1
                              UNTIL COLLEGE > 3
      *
                              AFTER DEPT FROM 1 BY 1
                                    UNTIL DEPT > 4.
      *
           DISPLAY 'SEMESTER TOTALS    ', SEMESTER-TOTALS-TABLE.
      *
       ACCUMULATE-SEMESTER-TOTALS.
           ADD ENROLLMENT (COLLEGE, DEPT, SEMESTER)
                  TO SEMESTER-TOTALS (SEMESTER).
           MOVE SPACES TO BLANK3 (SEMESTER).
      *
       TOTAL-ENROLLMENT-SUM.
           PERFORM ACCUMULATE-ENROLLMENT-TOTAL
                VARYING COLLEGE FROM 1 BY 1
                        UNTIL COLLEGE > 3
                   AFTER DEPT FROM 1 BY 1
                               UNTIL DEPT > 4
      *
                         AFTER SEMESTER FROM 1 BY 1
                                    UNTIL SEMESTER > 2.
           DISPLAY 'TOTAL ENROLLMENT      ', ENROLLMENT-TOTAL.
      *
       ACCUMULATE-ENROLLMENT-TOTAL.
           ADD ENROLLMENT (COLLEGE, DEPT, SEMESTER)
                               TO ENROLLMENT-TOTAL.
```

FIGURE 12-10 *(Continued)*
SAMPLE PROGRAM WITH A THREE-DIMENSIONAL TABLE.

Then there are four illustrative modules: COLLEGE-SUM, DEPARTMENT-SUM, SEMESTER-SUM, and TOTAL-ENROLLMENT-SUM. The first one, COLLEGE-SUM, forms the enrollment total for each of the three colleges in a WORKING-STORAGE table called COLLEGE-TOTALS-TABLE. Notice use of the VARYING . . . AFTER . . . AFTER construct. When the accumulation has been completed we simply DISPLAY the contents of COLLEGE-TOTALS-TABLE to simplify the program, since our objective is to illustrate the processing logic rather than the output format.

The second module, DEPARTMENT-SUM, specifies . . . VARYING DEPT . . . AFTER COLLEGE . . . AFTER SEMESTER. Contrast that to the PERFORM in COLLEGE-SUM and notice that since we are accessing all the data in the table, the order in which we vary the subscripts does not matter.

The other two modules, SEMESTER-SUM and TOTAL-ENROLLMENT-SUM, are similar in structure to the first two modules. It should be commented that there is known repetition in the program. Actually, one PERFORM . . . VARYING statement would be sufficient to accumulate all four totals by a paragraph such as the following:

```
ACCUMULATE-TOTALS.
    ADD ENROLLMENT (COLLEGE, DEPT, SEMESTER)
      TO COLLEGE-TOTALS (COLLEGE)
         DEPT-TOTALS (DEPT)
         SEMESTER-TOTALS (SEMESTER)
         ENROLLMENT-TOTAL.
```

The program in Figure 12-10 treats each accumulation task independently of the others to provide an illustration of how we could access all the data by each respective category, or dimension. Although the illustration has to do with accumulation of totals, the processing logic would remain essentially the same regardless of the specific task.

TABLE SEARCHING WITH COBOL

The use of a dictionary to look up the meaning of words is perhaps the most common example of the need to search a table or file for particular entries. In the context of data processing, searching tables to find particular entries stored in them is a fundamental process. Because of the frequency with which table searching is used in computer programs, the COBOL language includes a specialized set of instructions to facilitate the programming task. Suppose, for example, that a company maintains a list of office employees in alphabetical order. There are 400 employees, and each is assigned a four-digit employee identification number. Because of coding requirements, such as having the departmental identification included in the code, there is no correspondence between the numeric order of the identification numbers and the alphabetic order of employee names. The DATA DIVISION statements that describe the table can be written as follows:

```
01  DATA-TABLE.
    02  NAME-NUMBER
        OCCURS 400 TIMES.
        03  NAME        PICTURE X(16).
        03  EMPL-NO     PICTURE 9999.
```

In this section, we consider two data processing tasks. First, given the name of an employee, determine his or her employee-number. Second, given the employee-number, determine his or her name. The procedures to be used with both sorted and unsorted tables also are described.

One Method for Searching a Sorted Table

Assume that the table is sorted on the NAME field. Since the table will be searched by reference to the NAME field, the record description entry can be written in such a way as to indicate the fact that the table is sorted on NAME in ascending order. Further, the record description can be written so that reference to table entries can be made by use of a variant of the subscript concept, called an *index*. Thus, in place of the table description in the general DATA DIVISION statements cited above, the following statements can be written:

```
01  DATA-TABLE.
    02  NAME-NUMBER
        OCCURS 400 TIMES.
        ASCENDING KEY IS NAME
        INDEXED BY WHICH-ONE.
        03  NAME        PICTURE X(16).
        03  EMPL-NO     PICTURE 9999.
```

Notice the ASCENDING KEY IS statement, which indicates that the item defined by the OCCURS clause is sorted in ascending sequence. The INDEXED BY WHICH-ONE clause defines WHICH-ONE as an index for NAME-NUMBER. Unlike a subscript, an index is *not* defined or described by a DATA DIVISION entry and PICTURE clause. It is a numeric field, normally binary, whose length is defined by the compiler used. If we write NAME (WHICH-ONE), we thus reference an entry in the table just as with the use of a regular subscript. Unlike a subscript, however, the value of WHICH-ONE is defined by one of three verbs: SET, SEARCH, or PERFORM. Rather than elaborating further on the use of WHICH-ONE as an index, the following example is used to provide additional understanding of this option.

Suppose that EMPL-NAME is a field that contains a name for which we want

to determine the associated employee-number. In order to accomplish this task, we can execute the following PROCEDURE DIVISION statements:

```
SEARCH ALL NAME-NUMBER
    AT END PERFORM CANT-FIND
    WHEN NAME (WHICH-ONE) = EMPL-NAME
        MOVE EMPL-NO (WHICH-ONE) TO EDIT-NUMBER.
```

The instruction SEARCH ALL NAME-NUMBER indicates that the NAME-NUMBER table should be searched. Recall that in the DATA DIVISION the OCCURS clause defined ASCENDING KEY IS NAME; therefore, the search is conducted with respect to the NAME field. If no match is found, the AT END clause is executed. As the table is searched, the procedure is to vary WHICH-ONE and to compare the resulting NAME (WHICH-ONE) to EMPL-NAME. If the two match, the MOVE statement that follows the WHEN is executed next and the search terminates. If no match is found, WHICH-ONE is changed to a new value and the procedure is repeated until the entire table has been searched. How is the value of WHICH-ONE varied? The instructions are included in the compiler. Since the table in the present example is sorted, it is likely that the compiler will provide instructions for a binary search. If a linear search were used, the values of WHICH-ONE in this example would be 1, 2, 3, . . . , 399, 400. Again, note that the search procedure is supplied automatically by the compiler, so that the programmer need not write any of the detailed instructions for this search. When a match is found, program execution is transferred to the instruction MOVE EMPL-NO (WHICH-ONE) TO EDIT-NUMBER following the WHEN and the search terminates. Notice that WHICH-ONE has the reference value of the record for which the match occurred; thus, it is used as a subscript, or index, to reference the corresponding EMPL-NO.

An Alternative Approach for Sorted Tables

We now demonstrate some options, with respect to the preceding search procedure, by using an alternate approach to the task. First, we rewrite the DATA DIVISION entries as follows:

```
01 DATA-TABLE.
   02 NAME
          OCCURS 400 TIMES
          ASCENDING KEY IS NAME
          INDEXED BY NAME-INDEX
          PICTURE IS X(16).
   02 EMPL-NO
          OCCURS 400 TIMES
          INDEXED BY NO-INDEX
          PICTURE 9999.
```

In effect, two tables are defined, one called NAME and the other called EMPL-NO. The NAME table is sorted on ascending sequence and is indexed by NAME-INDEX, while the EMPL-NO table is indexed by NO-INDEX. Of course, the two PICTURE descriptions refer to the NAME and EMPL-NO fields, respectively, and *not* to the index names.

In relation to these DATA DIVISION statements, the following PROCEDURE DIVISION statements then can be written:

```
SET NAME-INDEX TO 1.
SET NO-INDEX TO 1.
SEARCH NAME VARYING NO-INDEX
    AT END PERFORM NO-MATCH
    WHEN EMPL-NAME = NAME (NAME-INDEX)
        MOVE EMPL-NO (NO-INDEX) TO EDIT-FIELD.
```

The SEARCH statement contains the VARYING option, which in effect indicates that NO-INDEX is to be varied in the same way as the index of the table being searched (in this case NAME-INDEX). Thus, for example, when NAME-INDEX equals 10, NO-INDEX also is equal to 10. As a result, the EMPL-NO (NO-INDEX) referred to in the MOVE statement corresponds to the NAME for which a match was found. If a match occurred in the thirty-fourth position of the NAME table, for example, NAME-INDEX and NO-INDEX both would have a value of 34. Thus, the employee-number that is output corresponds to the name of the employee for which a match was found. Since the ALL option is not used with the SEARCH verb, the search procedure involves accessing every entry in the table in sequence until a match is found or the entire table is searched. In effect, then, we do not take advantage of the fact that the table was sorted.

Searching an Unsorted Table

We now consider the search procedure that can be used with an unsorted table. Further, though it makes no difference in the general procedure used, we now shall assume that we have an employee-number and we wish to find the corresponding name, rather than vice versa. The relevant DATA DIVISION entries can be written as follows:

```
01  DATA-TABLE.
    02  NAME-NUMBER
        OCCURS 400 TIMES
        INDEXED BY N.
    03  NAME                PICTURE X(16).
    03  EMPL-NO             PICTURE 9999.
```

The NAME-NUMBER table is indexed by N, which is an index-name of our choice. Assume that the employee-number in question is stored in EMPLOYEE-NUMBER. In the PROCEDURE DIVISION, we can write:

```
SET N TO 1.
SEARCH NAME-NUMBER
    AT END PERFORM NO-MATCH
    WHEN EMPLOYEE-NUMBER = EMPL-NO (N)
        MOVE NAME (N) TO EDIT-NAME.
```

The value of N is initialized at 1 by the statement SET N TO 1. Since the table has not been sorted (and particularly, since it has not been sorted with respect to EMPL-NO), a linear search will be performed beginning with the first entry of the table. If we have a reason to begin the search at another point, either constant or variable, such statements as SET N TO 52 or SET N TO FIRST-VALUE can be written. Notice that the SEARCH verb does not include use of the ALL option. The ALL option can be used only with tables that have the KEY option in OCCURS; that is, it can be used only with sorted tables. The search procedure followed is to vary N from its first value of 1, by increments of 1, up to the last value (in this case 400), each time testing to determine if EMPLOYEE-NUMBER equals EMPL-NO (N). If a match is found, NAME (N) is moved to EDIT-NAME so that it can be output. If no match is found in the entire table, AT END is executed.

Review

1 The variant of the subscript concept, which can be used to reference table entries, is called a(n) ______________.

index

2 The value of an index used to reference table entries is defined by one of three verbs: SET, SEARCH, or PERFORM. If a SEARCH instruction is

executed and no match is found in the data table, the ______ clause is always executed.

AT END

3 By the alternative approach for determining the employee-number, given the name of the employee, two tables were defined, one for NAME and the other for EMPL-NO. The index number associated with EMPL-NO was set equal to the index of the NAME table being searched by the use of the ______ option.

VARYING

4 When an unsorted table is searched, the method always used is ______ search.

linear

COBOL LANGUAGE OPTIONS IN TABLE SEARCHING

Now that we have explored the basic ideas associated with table searching by means of examples, we can turn our attention to a formal consideration of the language options. We do not consider all the possible options that could be used; rather, we describe the basic ones.

The OCCURS Clause

First, let us consider the format associated with the OCCURS clause:

```
OCCURS integer TIMES

[ { ASCENDING  } KEY IS data-name-1 [data-name-2] . . . ]
[ { DESCENDING }                                        ]

[INDEXED BY index-name-1 [index-name-2] . . .]
```

Some examples may help illustrate the available forms of the OCCURS option. Let us consider this one first:

```
02  SAMPLE OCCURS 100 TIMES
    ASCENDING KEY IS YEAR MONTH.
    03  OTHER-DATA      PICTURE X(20).
    03  MONTH           PICTURE 99.
    03  YEAR            PICTURE 99.
    03  REST-OF-IT      PICTURE X(40).
```

In this example there are two keys, YEAR and MONTH. Keys are listed in decreasing order of significance. Thus, the months are sorted in ascending sequence within the years, which are also in ascending sequence. As the example illustrates, the order in which the keys appear in the KEY clause is *not* related to their physical order in the record.

Here is a second example:

```
02  SAMPLE PICTURE 9(8) OCCURS 100 TIMES
    INDEXED BY N.
```

In this case we specify an index called N, presumably for the purpose of later using this index in a SEARCH statement.

The USAGE Clause

Consider now the USAGE clause. We can specify that the USAGE of a data item is INDEX, so that the item can be used in conjunction with SET, SEARCH, or PERFORM statements. You may recall that USAGE IS DISPLAY and USAGE IS COMPUTATIONAL are the other available options, as shown in the following format:

```
         { DISPLAY       }
USAGE IS { COMPUTATIONAL }
         { INDEX         }
```

As an example of implementing the USAGE clause, consider the following program segment:

```
01  SAMPLE.
    02  FIRST-PART       PICTURE X(10).
    02  K USAGE IS INDEX.
```

The item called K is an INDEX item; therefore, no PICTURE clause is given. All index items are handled according to the rules associated with particular computer systems. Normally, index items are in binary form.

The SEARCH Verb

Consider now the SEARCH verb, which is the cornerstone of a search instruction. Two principal formats are available, as presented in Figure 12-11.

In Format 1, identifier-1 is an item whose description in the DATA DIVISION contains an OCCURS and an INDEXED BY clause. When the VARYING option is used, index-name-2 or identifier-2 is varied in the same way as the index specified in the relevant INDEXED BY clause. Identifier-2 must be specified as USAGE IS INDEX, or it must be an elementary integer item. The AT END clause is optional. If it is omitted, program control will pass to the next sentence when the table has been searched and no match has been found. If AT END is included and imperative-statement-1 does not terminate with GO TO, a branch to the next sentence will be made (in effect bypassing the WHEN clauses). WHEN introduces another form of conditional expression.

Format 2 is used with sorted tables, that is, tables for which the OCCURS clause contains a KEY in addition to the INDEXED BY option. The search may be a binary search, or any other method included in a particular compiler; however, as far as the programmer is concerned, only the instructions included in Format 2 are required.

Notice that in Format 2 only one WHEN option is available, but multiple AND conditions are allowed. Thus, all the conditions must be true in order for the search to be satisfied. In contrast, whenever multiple WHEN statements are used in Format 1, any one of these conditions being true constitutes a sufficient reason for search termination.

The SET Verb

Basically, the SET verb is a variation of MOVE and provides a way of handling index items. Two formats are available, as follows:

FORMAT 1

```
    { index-name-1 [index-name-2] . . . }      { index-name-3 }
SET {                                   }  TO  { identifier-3 }
    { identifier-1 [identifier-2] . . . }      { integer-1    }
```

FORMAT 2

```
SET index-name-1 [index-name-2] . . . { UP BY   } { identifier-1 }
                                      { DOWN BY } { integer-1    }
```

In Format 1, if we use index-name-1, we can set it equal to index-name-3, identifier-3, or literal-1. If identifier-3 is used, it must be defined as an elementary integer item; if integer-1 is used, it must be a positive integer. If we set identifier-1 and it has not been defined by a USAGE IS INDEX clause, it can be set only to index-name-3.

If we write SET A TO B, we basically are moving the value of B to A. In this way, we can transfer data to an index item or an identifier, for use either in a SEARCH or after a SEARCH. For instance, we may have two tables, one INDEXED BY A and the other INDEXED BY B. After performing a search and finding a match on the first table, we may want to reference the corresponding entry in the second table. We then SET B TO A so that B can be used as a subscript or index in the second table.

FORMAT 1

```
SEARCH identifier-1 [ VARYING { index-name-1 } ]
                              { identifier-2 }

[AT END imperative-statement-1]

WHEN condition-1 { imperative-statement-2 }
                 { NEXT SENTENCE          }

[ WHEN condition-2 { imperative-statement-2 } ] . . .
[                  { NEXT SENTENCE          } ]

[END-SEARCH]
```

FORMAT 2

```
SEARCH ALL identifier-1 [AT END imperative-statement-1]

      { data-name-1 { IS EQUAL TO } { identifier-2              } }
WHEN  {             { IS =        } { literal-1                 } }
      {                             { arithmetic-expression-1   } }
      { condition-name-1                                          }

[      { data-name-2 { IS EQUAL TO } { identifier-3             } } ]
[ AND  {             { IS =        } { literal-2                } } ] . . .
[      {                             { arithmetic-expression-2  } } ]
[      { condition-name-2                                         } ]

{ imperative-statement-2 }
{ NEXT SENTENCE          }

[END-SEARCH]
```

FIGURE 12-11
FORMATS OF THE SEARCH VERB.

In FORMAT 2, we can increase or decrease the value of index-name-1 either by a positive integer (literal-1) or by the value of identifier-1, which must be a field that has not been defined as USAGE IS INDEX. The effect of UP BY is to increase the value of index-name-1, while the effect of DOWN BY is to change index-name-1 by the indicated decrement.

Review

1 The COBOL language option used to indicate the total number of table entries and to identify the key or keys associated with the records is the ________ clause.

OCCURS

2 The COBOL option used to identify a particular data item as being an index is the ________________ clause.

USAGE

3 The COBOL verb that identifies the table to be searched and also includes options to indicate what should be done when a match is found as well as when it is not found is ________________.

SEARCH

4 The COBOL verb that is a variation of the MOVE verb and provides the basis for designating the value to be assigned to an INDEX is __________.

SET

SAMPLE PROGRAM WITH INDEXING AND SEARCHING

An input file consists of records containing a student name and the numeric score for each of eight parts of a standardized test. Our objective is to store the data in a table and then to illustrate use of SEARCH to identify those students who either scored 100.0 on any part of the test, or who scored less than 50.0 on any part of the test.

The program is presented in Figure 12-12. Notice that S is designated in the WORKING-STORAGE in INDEXED BY S, and then it is used in the PERFORM . . . VARYING of the PROGRAM-SUMMARY paragraph. Then observe that, after the PERFORM statement, we write SET S DOWN BY 1, which means to subtract 1 from S. After so adjusting S, it represents the number of student records that were read in. We then save the value of S in M (SET M TO S) to be used to control the printing task. Notice that M is a WORKING-STORAGE USAGE INDEX item.

After the table has been stored, we search it in the CHECK-SCORES-AND-PRINT paragraph. Notice that we did not include the AT END option in this use of Format 1 of the SEARCH, since it was not needed. If neither a score of 100 nor a score of less than 50 is found for a given student, we simply want to continue with the next student without doing anything AT END.

When either one of the two search conditions is true, we PERFORM PRINT-SCORES and the search for that student record terminates. Then, because of the PERFORM CHECK-SCORES-AND-PRINT in the PROGRAM-SUMMARY, we repeat the search for the next student.

The MOVE-SCORES paragraph illustrates again the use of SET. Since SCORE-OUT was indexed by R, we SET R TO Q so that in the MOVE statement we reference the corresponding scores.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. GRADES.
*
 ENVIRONMENT DIVISION.
*    CONFIGURATION SECTION.
 SOURCE-COMPUTER.  ABC-480.
 OBJECT-COMPUTER.  ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT EXAM-FILE    ASSIGN TO READER.
     SELECT REPORT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  EXAM-FILE         LABEL RECORDS OMITTED
                       DATA RECORD IS EXAM-REC.
 01  EXAM-REC.
     02  STUDENT-NAME                           PIC X(20).
     02  EXAM-SCORES.
         03 SCORES-IN  OCCURS 8 TIMES           PIC 9(3)V9.
     02  FILLER                                 PIC X(28).
*
 FD  REPORT-FILE       LABEL RECORDS OMITTED
                       DATA RECORD IS REPORT-REC.
 01  REPORT-REC                                 PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-SWITCH                     PIC XXX VALUE 'NO '.
     88  END-OF-FILE                        VALUE 'YES'.
*
 01  M   USAGE INDEX.
*
 01  EXAM-TABLE.
*
     02  STUDENT-DATA    OCCURS 50 TIMES
                         INDEXED BY  S.
*
         03  STUDENT-NAME                   PIC X(20).
*
         03  EXAM-SCORES.
*
             04  SCORE   OCCURS 8 TIMES PIC 9(3)V9
                         INDEXED BY Q.
*
 01  HEADING-1.
     02  FILLER                    PIC X(50) VALUE SPACES.
     02  FILLER                    PIC X(9)  VALUE 'EXAM PART'.
 01  HEADING-2.
     02  FILLER                    PIC X(8)   VALUE SPACES.
     02  FILLER                    PIC X(12) VALUE 'STUDENT NAME'.
     02  FILLER                    PIC X(12) VALUE SPACES.
     02  FILLER                    PIC X(50) VALUE
         '1         2         3         4         5         6         7         8'.
*
 01  REPORT-LINE.
     02  FILLER                    PIC X(5)   VALUE SPACES.
     02  STUDENT-NAME              PIC X(20).
     02  FILLER                    PIC X(2)   VALUE SPACES.
*
     02  SCORES-OUT          OCCURS 8 TIMES    PIC ZZZ99.9.
                             INDEXED BY R.
```

FIGURE 12-12

LISTING FOR PROGRAM WITH INDEXING AND SEARCHING.

```
*
/
 PROCEDURE DIVISION.
*
 PROGRAM-SUMMARY.
     OPEN  INPUT   EXAM-FILE
           OUTPUT  REPORT-FILE.
*
     PERFORM READ-EXAM-REC
*
     PERFORM STORE-RECORD
             VARYING  S  FROM  1  BY  1
                       UNTIL  S > 50
                       OR END-OF-FILE.
*
     SET  S   DOWN  BY  1.
     SET  M   TO  S.
*
     WRITE REPORT-REC FROM HEADING-1 AFTER PAGE
     WRITE REPORT-REC FROM HEADING-2 AFTER 2.
*
     PERFORM CHECK-SCORES-AND-PRINT
             VARYING  S  FROM  1  BY  1
                       UNTIL S > M.
*
     CLOSE EXAM-FILE  REPORT-FILE
*
     STOP RUN.
*
 READ-EXAM-REC.
     READ EXAM-FILE RECORD
          AT END MOVE 'YES' TO END-OF-FILE-SWITCH.
 STORE-RECORD.
     MOVE STUDENT-NAME OF EXAM-REC
          TP STUDENT-NAME OF EXAM-TABLE (S)
*
     MOVE EXAM-SCORES OF EXAM-REC
          TO EXAM-SCORES OF EXAM-TABLE (S).
*
     PERFORM READ-EXAM-REC.
 CHECK-SCORES-AND-PRINT.
     SET Q TO 1
     SEARCH SCORE
            WHEN   SCORE (S, Q) = 100.0
                   PERFORM PRINT-SCORES
*
            WHEN   SCORE (S, Q) < 50.0
                   PERFORM PRINT-SCORES.
*
 PRINT-SCORES.
     PERFORM MOVE-SCORES
             VARYING  Q  FROM  1  BY  1
                       UNTIL Q > 8.
*
     MOVE STUDENT-NAME OF EXAM-TABLE (S)
          TO SUDENT-NAME OF REPORT-LINE
     WRITE REPORT-REC FROM REPORT-LINE AFTER 2 LINES.
*
 MOVE-SCORES.
     SET R TO Q
     MOVE SCORE (S, Q) TO SCORE-OUT (R).
```

FIGURE 12-12 (*Continued*)
LISTING FOR PROGRAM WITH INDEXING AND SEARCHING.

Figure 12-13 illustrates sample input and output for the program. The program provides a somewhat contrived set of cases to illustrate the options. Exercise 12.5 at the end of the chapter asks the student to modify the example in order to incorporate other features of searching.

Sample Input

STUDENT-1	09051000082306000900089909770777
STUDENT-2	02000732066605920812085503000600
STUDENT-3	09000820100008320777075306890920
STUDENT-4	07450789082306950765074806780802
STUDENT-5	04500720073010000235024506250213

Sample Output

	EXAM PART							
STUDENT NAME	1	2	3	4	5	6	7	8
STUDENT-1	90.5	100.0	82.3	60.0	90.0	89.9	87.7	77.7
STUDENT-2	20.0	73.2	66.6	59.2	81.2	85.5	30.0	60.0
STUDENT-3	90.0	82.0	100.0	83.2	77.7	75.3	68.9	92.0
STUDENT-4	74.5	78.9	82.3	69.5	76.5	74.8	67.8	80.2
STUDENT-5	45.0	72.0	73.0	100.0	23.5	24.5	62.5	21.3

FIGURE 12-13
SAMPLE INPUT AND OUTPUT FOR THE PROGRAM WITH INDEXING AND SEARCHING.

EXERCISES

12.1 A marketing survey conducted by a company involved administering a questionnaire of 25 questions. The responses to each question have been coded by a 1-digit code, ranging from 0 to 9. We want to accumulate a table of the responses to each of the 25 questions, as shown in Figure 12-14. Write DATA DIVISION entries to form such a table. It should be possible to make reference to each individual cell in the table, as well as each row (question).

12.2 Suppose that the table described in the following contains 20 values:

```
01 SALES-DATA-TABLE.
   02 SALES-TERRITORY
      OCCURS 5 TIMES.
      03 QUARTER-SALES
         OCCURS 4 TIMES
         PICTURE 9(6)V99.
```

We wish to print the table values with the rows representing quarters and the columns representing sales territories. Write the necessary program instructions.

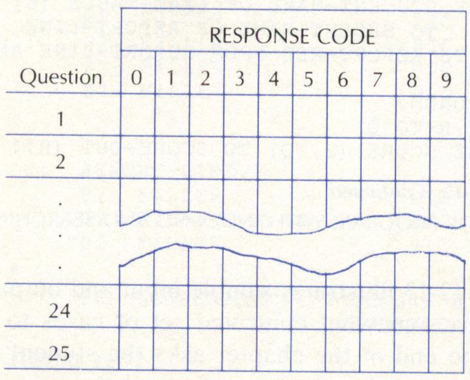


FIGURE 12-14
REQUIRED TABLE FOR EXERCISE 12.1.

ACME Corporation
Sales Analysis Report

MONTH	WEST	MIDWEST	SOUTH	EAST	MONTHLY TOTAL	PERCENT OF YEAR TOTAL
JANUARY						
FEBRUARY						
MARCH						
APRIL						
MAY						
JUNE						
JULY						
AUGUST						
SEPTEMBER						
OCTOBER						
NOVEMBER						
DECEMBER						
TOTAL						
PERCENT OF TOTAL						

FIGURE 12-15
REQUIRED SALES REPORT FOR EXERCISE 12.3.

12.3 Sales records contain data as follows:

COLUMNS	DATA
1–5	Amount (whole dollar)
6	Region code: 1 = West, 2 = Midwest 3 = South, 4 = East
7–8	Month code (numeric)

We want to be able to read in the data and produce the sales analysis report illustrated in Figure 12-15. Input records that contain error region codes, error month codes, or nonnumeric amounts should be excluded from the tabulation and are to be printed with a suitable error message. Data are presented in Figure 12-16. Write a program to produce such a report.

12.4 A Source file contains data of the following form:

COLUMNS	FIELD
1	Store i.d. (M = Metro, F = Fiesta)
2	Day of the week (1 = Sunday, 2 = Monday, etc.)
3–4	Month (1 = January, etc.)
5–7	Amount of sale (no decimal places)

Assume that the data have been edited so there is no need to check for errors. Write your program with the assumption that all the data are correct.

The program should read each source record, determine its classification, and add the amount of the sale to the corresponding location in the table.

00133412	00197308	00152201	00160108	00060412
00096410	00142409	00118408	00152109	00030411
00085201	00356407	00190406	00202100	0015412
00099101	00428407	00150309	00200312	00018311
00110202	00234202	00160110	00145311	00072312
00100303	00152103	00213211	20200410	00035211
00050405	00123101	00115112	00900309	00150303
00040004	01100205	00118109	00100106	00160202
00030704	01200307	00219208	00500206	00110101
00018103	00142211	00300402	/ 560507	00200204
00025302	00156208	00200203	70082 01	00220308
00020401	00171307	00145302	00233102	00183407
00010106	00145405	00142304	00422109	00157301
00123102	00201403	00140205	00150212	00085409
00185303	00300302	00150207	00050211	00096110

FIGURE 12-16

INPUT DATA FOR EXERCISE 12.3.

Note that the source records are not in any particular order and that there could be more than one record for a given store, day of the week, and month. For example, there could be two records for the Metro store on Tuesday(s) in January.

As output, the program should produce a table such as the one presented in Figure 12-17. Notice that monthly sales are grouped together into quarters of the year. Some data are included in the table in order to clarify the meaning of the totals and percentages required. For example, the 2 percent at the right side of the SUNDAY row is computed by dividing the total Sunday sales at the Fiesta store (60) by the total sales of the year (3000) and converting to a percentage.

12.5 Modify the sample searching program in the last section of the chapter as follows. Assume that the student names are sorted in ascending sequence. After the table is created, read records, each of which contains a student name. Search the table and print the test score data for that student. If the name of the student cannot be found, print a suitable error message.

American Sales Corporation

Annual Sales by Quarter and Day of the Week

DAY	JAN–MAR		APR–JUN		JUL–SEP		OCT–DEC		DAILY TOTAL			% OF YR. TOTAL	
	METRO	FIESTA	METRO	FIESTA	METRO	FIESTA	METRO	FIESTA	METRO	FIESTA	TOTAL	METRO	FIESTA
SUNDAY	50	30	20	0	30	10	40	20	140	60	200	5%	2%
MONDAY													
.													
.													
.													
SATURDAY													
TOTAL	120	80							1000	2000	3000		
% OF QUARTER TOTAL	60%	40%							33%	66%	100%		

FIGURE 12-17

REQUIRED OUTPUT FOR EXERCISE 12.4.

12.6 Write a COBOL program segment to perform a search in a sorted table called TABLE, searching for an entry that matches THIS-RECORD. If a match is found, the program goes to the paragraph MATCHED; otherwise, it goes to the paragraph NO-MATCH. Use I to hold the position of the matching table entry, if there is a match. Assume the following data descriptions:

```
01  THIS-RECORD        PICTURE X(12).
01  I                  PICTURE 999.
01  TABLE.
    02  CELL           PICTURE X(12) OCCURS 100 TIMES
                       INDEXED BY N.
```

12.7 How does COBOL differentiate between searching sorted and unsorted tables?

12.8 What is the difference between an index and a subscript in COBOL?

12.9 Review the meaning of the following search-related COBOL features, explaining the use of each feature:

- **a** OCCURS accompanied by the ASCENDING, (DESCENDING) KEY options
- **b** INDEXED BY
- **c** USAGE IS INDEX
- **d** SEARCH and its several optional forms
- **e** SET TO and SET UP BY or SET DOWN BY

12.10 Incorporate the indexing capability of COBOL into a program that you have written previously, and use the SEARCH verb in conjunction with any required search.

13 INDEXED SEQUENTIAL FILE PROCESSING

■ INTRODUCTION

In Chapter 9 we introduced the three common methods of file organization: *sequential, indexed sequential,* and *relative,* or *direct*. We also described programming concepts and methods specifically concerned with the sequential method of file organization. In this chapter we describe indexed sequential files, frequently referred to as Indexed Sequential Access Method, or ISAM, files. In the following chapter, the processing of relative files is described.

■ INDEXED SEQUENTIAL FILE ORGANIZATION

This form of file organization represents something of a balance between sequential file organization and relative file organization. It allows sequential storage but facilitates random accessing or processing. The method utilizes an index table that indicates the approximate storage location for a given record. An indexed sequential file consists of two principal components: the *main file* and the *index*.

The *main file* is the storage area that includes the records in sequential order. It consists of two segments, namely, the *prime* area and the *overflow* area. The prime area is loaded with the records in sequential order when the file is first created, whereas the overflow area is designed to accommodate additions to the file.

The *index* (or index file) is designed to facilitate random access to records in the main file. Its function is similar to the labels attached to the outside of each card drawer in a typical library card catalog. If we are trying to locate the card reference to an author, say, C. A. Johnson, we consult the labels on the fronts of the card drawers. We start from the beginning until we find the drawer that

indicates that the last card in that drawer exceeds "Johnson" or, by coincidence, that it is "Johnson." In a somewhat similar way, the index file consists of records referencing the last record in each group of records in the main file. When we want to access a record identified by a key, such as "Johnson," the index is searched until we find a record that equals or exceeds "Johnson." At that point, we have identified the group of records that includes the "Johnson" record.

Any number of indexing methods are used. We describe here one method as an illustration. Because disks are subdivided naturally into cylinders and tracks within cylinders, the index file often consists of two levels of indexes, as illustrated in Figure 13-1. In the figure, we have assumed a simplified case of a file written on three cylinders, each cylinder consisting of 10 tracks (numbered 00 through 09). The first record in the cylinder index shows that the last record in that cylinder has a key of DAN, and that the track index for that cylinder is in cylinder 0, track 00. Turning to the track index for that cylinder, we see that the last record on the first track (cylinder 0, track 00) is ANDY, the last record on the second track is CHRIS, and the last record on the tenth track is DAN. Suppose we are trying to locate the record for GIGI. The process is as follows. First, the cylinder index is searched until LINDA is encountered. Since LINDA is higher than GIGI, the GIGI record must be in the second cylinder (cylinder 1). Next, we find that the first record on the track index for cylinder 1 is GEORGE; so we read the next index record—HELEN. Now we know that GIGI is in the second track (track 1) of the second cylinder. At this point, the identified track is read sequentially to locate the record itself (or perhaps to discover that there is no record identified as GIGI).

This illustration highlights the structure and use of the index file in an indexed

Cylinder Index

CYLINDER NUMBER	HIGHEST KEY IN CYLINDER	ADDRESS OF TRACK INDEX FOR CYLINDER
0	DAN	00
1	LINDA	10
2	WILLIE	20

Track Indexes for Each Cylinder

TRACK ADDRESS OF TRACK INDEX	HIGHEST KEY IN TRACK	ADDRESS OF TRACK	
00	ANDY	00	Cylinder 0
00	CHRIS	01	
	.		
	.		
00	DAN	09	
10	GEORGE	10	Cylinder 1
10	HELEN	11	
	.		
	.		
10	LINDA	19	
20	MARIAN	20	Cylinder 2
20	PAT	21	
	.		
	.		
20	WILLIE	29	

FIGURE 13-1
ILLUSTRATION OF CYLINDER AND TRACK INDEXES.

sequential file. Creation of the index is done by the operating system, not by the applications programmer; nevertheless, it is useful to understand the rudiments of the indexing method. The method used in the illustration is one of several ways that are used to create index files. Another popular method used with many large IBM mainframe computers is known as VSAM, which stands for Virtual Storage Access Method. However, this method is considerably more complex than the illustration above, and its description is outside of the intended scope of this book.

When a file is created initially as an indexed file the records must be in ascending order with respect to their key, and the operating system creates both the main file and the index. The area on which the main file is written during its creation is called the *prime* area. After the file is created initially, we may want to add records to it. This is permissible, but it introduces some complications. The added records result in creation of *overflow,* which is an area separate from the prime area. Thus, *physically,* the main file no longer is sequential; however, from the point of view of the programmer, the file is *logically* sequential. The records in the prime tracks are linked to the records in the overflow tracks, so that the file is accessible as if it were in physical sequential order. We also should point out that, just as the main file is subdivided into prime and overflow areas, so is the index file enhanced to include index records in prime and in overflow tracks. The details of overflow concepts and their implementation are outside the scope of this book and are specific to the indexing method employed by the operating system. Still, the user of COBOL should be aware of the fact that the presence of overflow slows down the processing of an indexed file. As a result, when "enough" records have been added to degrade performance sufficiently, the file is re-created to organize it physically as one file in prime areas and to eliminate the overflow.

When records are deleted from the file, they normally are "tagged" as deleted, but they are not physically erased. Again, from the viewpoint of the applications programmer, the records are not there, since the operating system "skips" over them as the result of reading the deletion code written on the record.

A further consideration with respect to indexed organization is that it is possible to have more than one index file for a given main file. In this context we speak of the *primary* and one or more *secondary* keys. For example, we may create a customer file based on customer name as the primary key. In addition we may want to be able to access customer records based on the customer's telephone number. In such a case we would create a secondary key index based on telephone number. In COBOL, secondary keys are referred to as ALTERNATE keys, as you will observe later in this chapter, but *secondary key* is the generally accepted term.

The primary key value for each record is the basis for the physical positioning of records in the file and must be unique. Secondary keys may be nonunique and do not affect the physical order of records. For example, if customer name is the primary key, the records would be stored in the order of the customer names. Assuming telephone number to be a secondary key, the records could not be in order of telephone number since they are in fact stored in order of name. Further, two or more customers, such as members of the same family having separate accounts, may have the same (nonunique) telephone number. However each customer must have a unique name, so that records can be identified unambiguously.

Review

1 The type of disk file organization that allows sequential storage but facilitates random accessing or processing is called ________________ file organization.

indexed sequential

2 The *main file* in an indexed sequential file includes two types of file storage areas referred to as the ______________ area and the ______________ area.

prime; overflow

3 The index file used in conjunction with an indexed sequential file often consists of two levels of indexes: a ______________ index and a ______________ index.

cylinder; track

4 Suppose that a file is written on five cylinders of 10 tracks each and that the cylinder index indicates that a particular record is in cylinder 3. The track location for this record then is determined by reference to track ________ of cylinder ________.

0; 3 (because the track index is commonly written in the first track of the designated cylinder)

5 After records are added to an indexed sequential file, the added records are stored in the [prime / overflow] area.

overflow

6 After records are added to an indexed sequential file, the file no longer is sequential [logically / physically] but is sequential [logically / physically].

physically; logically

7 Records that are deleted from an indexed sequential file normally [are / are not] physically erased from the file.

are not

8 When there is more than one index file for a main file, the one that is the basis for the physical positioning of the records in the file is the ______________ key, while the other index file(s) is / are referred to as the ______________ key(s).

primary; secondary

9 Every file [must / need not] have a unique primary key and any secondary keys [must / need not] be unique.

must; need not

AN EXAMPLE OF THE CREATION OF AN INDEXED FILE

An indexed file can be created with minimal effort on the part of the programmer. The source records first must be sorted in ascending sequence on a data field that will serve as the primary key for the file. Because an indexed file is a sequential file, the records are positioned in ascending order. We now consider an example to illustrate the process.

Suppose that we have source records that contain data about vendors. Columns 1–8 contain an identifier that is unique for each vendor and columns 9–80 contain other relevant data. We want to copy these records on disk, forming an indexed file sorted on the basis of the data in columns 1–8. Each disk record will consist of 68 characters.

First, we sort the source records on columns 1–8. In the following sample data, the records already are sorted, except for the fifth record (37654310), which is placed out of order purposely to illustrate the effects of such a condition:

```
12345678ACME-10
22345678ACME-20
35120001ACME-30
37890123ACME-40
37654310ACME-33
43215678ACME-50
49678912ACME-55
50000000ACME-60
60000000ACME-70
61234567ACME-80
70000000ACME-88
88888888ACME-89
90000000ACME-90
99999999ACME-99
```

Figure 13-2 presents a COBOL program written to create an indexed file. Notice the SELECT statement. ACCESS IS SEQUENTIAL specifies that the access mode for this file is sequential. ORGANIZATION IS INDEXED specifies that this is an indexed file. RECORD KEY IS VENDOR-NUMBER specifies that there is a field called VENDOR-NUMBER that is a field in the file record and that serves as the primary key for the file. In other words, the file will be in ascending order of VENDOR-NUMBER values.

In the DATA DIVISION included in Figure 13-2, observe that VENDOR-NUMBER is a field in the record description of the indexed file (VENDOR-FILE). The RECORD KEY must be a field in the file record, and it must be an alphanumeric field.

In the PROCEDURE DIVISION of Figure 13-2, the relevant part is:

```
FILE-CREATE.
        MOVE VENDOR-INDENT TO VENDOR-NUMBER
        MOVE VENDOR-DATA OF SOURCE-RECORD
          TO VENDOR-DATA OF VENDOR-RECORD
        WRITE VENDOR-RECORD
            INVALID KEY PERFORM ERROR-CASE.
        PERFORM READ-SOURCE-FILE.
```

We move VENDOR-INDENT, a field in the source record, to VENDOR-NUMBER, which was declared to be the RECORD KEY in the SELECT statement. The move of VENDOR-DATA simply transfers the other fields of the source record to the output record. The WRITE statement now includes the INVALID KEY condition. This condition is true whenever the record key of the record about to be written is not greater than the key of the preceding record in the file. If the INVALID KEY condition is true, the imperative statement is executed—in this case, PERFORM ERROR-CASE. Reviewing ERROR-CASE in Figure 13-2, we observe that an error message and the record with the error

```
IDENTIFICATION DIVISION.
PROGRAM-ID. INDEXFILE.
*
ENVIRONMENT DIVISION.
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
INPUT-OUTPUT SECTION.
*
FILE-CONTROL.
    SELECT VENDOR-FILE ASSIGN TO OLDMSTR
                  ACCESS IS SEQUENTIAL
                  ORGANIZATION IS INDEXED
                  RECORD KEY IS VENDOR-NUMBER.
*
    SELECT SOURCE-FILE ASSIGN TO READER.
```

FIGURE 13-2
A COBOL PROGRAM TO CREATE AN INDEXED FILE.

```
*
       SELECT PRINT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
 FD   VENDOR-FILE LABEL RECORDS ARE STANDARD
                DATA RECORD IS VENDOR-RECORD.
*
 01   VENDOR-RECORD.
      02 VENDOR-NUMBER            PIC 9(8).
      02 VENDOR-DATA              PIC X(60).
*
 FD   SOURCE-FILE LABEL RECORDS ARE STANDARD
                DATA RECORD IS SOURCE-RECORD.
*
 01   SOURCE-RECORD.
      02 VENDOR-IDENT                   PIC 9(8).
      02 VENDOR-DATA                    PIC X(72).
*
 FD   PRINT-FILE LABEL RECORDS IS OMITTED
                DATA RECORD IS PRINT-RECORD.
 01   PRINT-RECORD               PIC X(132).
*
 WORKING-STORAGE SECTION.
 01   END-OF-DATA-INDICATOR     PIC 9 VALUE ZERO.
      88 INPUT-ENDED    VALUE 1.
*
/
 PROCEDURE DIVISION.
*
 MAIN-ROUTINE.
      OPEN INPUT SOURCE-FILE
           OUTPUT VENDOR-FILE, PRINT-FILE.
      PERFORM READ-SOURCE-FILE.
*
      PERFORM FILE-CREATE UNTIL INPUT-ENDED.
*
      CLOSE SOURCE-FILE, VENDOR-FILE, PRINT-FILE.
      STOP RUN.
 READ-SOURCE-FILE.
      READ SOURCE-FILE RECORD
           AT END MOVE 1 TO END-OF-DATA-INDICATOR.
*
 FILE-CREATE.
      MOVE VENDOR-IDENT TO VENDOR-NUMBER
      MOVE VENDOR-DATA OF SOURCE-RECORD
           TO VENDOR-DATA OF VENDOR-RECORD
      WRITE VENDOR-RECORD
           INVALID KEY PERFORM ERROR-CASE.
      PERFORM READ-SOURCE-FILE.
*
 ERROR-CASE.
      MOVE ' INVALID KEY CONDITION FOR THIS RECORD'
            TO PRINT-RECORD
      WRITE PRINT-RECORD AFTER ADVANCING 1 LINE
      WRITE PRINT-RECORD FROM SOURCE-RECORD AFTER 1 LINE.
```

FIGURE 13-2 *(Continued)*
A COBOL PROGRAM TO CREATE AN INDEXED FILE.

are printed. Using the sample input data presented earlier, we would receive one error message and the fourth record would be printed, since it is the one record out of sequence.

Creation of an indexed file is a complex task, yet the language is very high level with respect to this task. The programmer need write very few instructions to invoke the procedure necessary for the task. In review, these instructions involve a few clauses in the SELECT statement, provision for a record key in the record description of the file, and moving data to the output record in the PROCEDURE DIVISION. In the following section, we study these specialized instructions in a more thorough and comprehensive framework.

Review

1 An indexed file essentially is a [sequential / direct] file.

sequential

2 In the ENVIRONMENT DIVISION, after the file is described as being SEQUENTIAL and INDEXED, the basis on which the file is sorted is identified by the COBOL reserved words ________________.

RECORD KEY

3 The RECORD KEY field must be [alphabetic / alphanumeric] and it [must / need not] be a field in the file record.

alphanumeric; must

4 The fact that one or more records to be written in an indexed file are not in the appropriate sequence is detected and identified by using the ________ option in conjunction with the WRITE statement.

INVALID KEY

COBOL LANGUAGE INSTRUCTIONS FOR INDEXED FILES

There are two divisions of a COBOL program that involve special instructions for indexed sequential files—the ENVIRONMENT DIVISION and the PROCEDURE DIVISION.

ENVIRONMENT DIVISION

Figure 13-3 presents the general format for the SELECT statement. Note that in the ASSIGN portion of the statement the revised version of COBOL provides for a literal option as a means for referencing the physical file, just as is true for the sequential file format, as explained in Chapter 9.

The ORGANIZATION statement specifies that this is an indexed file.

The ACCESS MODE clause specifies the way records in the file will be accessed. ACCESS MODE IS SEQUENTIAL specifies that records will be accessed in ascending order of the record key. Omission of the ACCESS clause defaults to the SEQUENTIAL option. Thus, up to now, our omission of this clause from programming considerations in this book has implied the sequential access mode. The RANDOM option specifies that the order in which records are

```
SELECT file-name-1

    ASSIGN TO { implementor-name }
              { literal          }
    [ORGANIZATION IS]   INDEXED

    [                  { SEQUENTIAL } ]
    [ ACCESS MODE IS   { RANDOM     } ]
    [                  { DYNAMIC    } ]

    RECORD KEY IS data-name-1
    [ALTERNATE RECORD KEY IS data-name-2
         [WITH DUPLICATES]] . . .
    [FILE STATUS IS data-name-3]
```

FIGURE 13-3
GENERAL FORMAT FOR THE SELECT STATEMENT FOR INDEXED FILES.

accessed will be controlled by the programmer. This control is accomplished by moving the value of the key of the desired record into the RECORD KEY field and then issuing an input-output command (READ, WRITE, REWRITE, DELETE).

The DYNAMIC option allows the programmer to change at will from sequential access to random access, using appropriate forms of input-output statements. In its absence, the file for a given program must be declared to be either in SEQUENTIAL or in RANDOM access mode, but not in both modes in the same program.

We repeat here a point made in the previous section on creating an indexed file. When the file is first being created, it *must* be in sequential access mode. In subsequent uses, it may be in any of the three options, SEQUENTIAL, RANDOM, or DYNAMIC.

RECORD KEY references a data-name that must be a field within the record description of the file. In case of multiple records, a field from any record description may be given. RECORD KEY specifies the primary key field, on the basis of which the file is sorted. The ALTERNATE RECORD option specifies a secondary key that is an alternate record key for the file. When alternate keys are used, we can access records either on the basis of the *primary* key specified in the RECORD KEY clause or on the basis of another ALTERNATE RECORD KEY. The file, of course, is always sorted on the basis of the primary record key. The DUPLICATES phrase specifies that the value of the associated alternate record key may be duplicated within any of the records in the file. In the absence of this phrase, the presence of duplicate key values is an error condition. Notice that duplicate key values are permitted for alternate keys. Each record must have a unique primary key. An example of alternate key values that are duplicates may be the ZIP code in a record where the ZIP code was specified as an alternate record key. Such a key allows the accessing of a record with a specified ZIP code—with recognition that the record so accessed will not be unique with respect to this (alternate) key value.

The FILE STATUS option operates just as explained for sequential files in Chapter 9. Figure 13-4 presents some selected status codes and their explanation.

SELECTED FILE STATUS CODES	EXPLANATION
00	Successful completion.
02	Successful completion but a duplicate key is detected.
04	A READ statement was executed but the length of the record is inconsistent with the record description.
21	Sequence error exists. The program changed the primary key between a READ and a REWRITE statement.
23	Attempt to read a nonexistent record.
24	An attempt is made to write beyond the boundaries of the file, such as when the file is full.
41	Attempt to OPEN file that is already open.
42	Attempt to CLOSE a file that has not been opened.
43	Attempt to DELETE or REWRITE without the prior execution of a corresponding successful READ.
47	Attempt to READ or START on a file not open in the INPUT or I/O mode.
49	Attempt to DELETE or REWRITE on a file not open in the I/O mode.

FIGURE 13-4
SELECTED FILE STATUS CODES.

For a complete set of codes it is advisable to refer to your own operating system, since many codes are specific to the implementor.

Review

1 In the SELECT statement in the ENVIRONMENT DIVISION, the fact that a file is to be set up as an indexed file is specified by the ________ statement.

ORGANIZATION

2 A file organized as an indexed file [can / cannot] be accessed in a sequential manner.

can

3 Omission of the ACCESS clause in the SELECT statement for an indexed file results in the file having to be accessed by the [sequential / random] mode.

sequential

4 The ACCESS MODE option that allows the programmer to change at will from sequential access to random access is called the ________ mode.

DYNAMIC

5 When the ALTERNATE RECORD KEY is used to access a record in a file, the file has to be sorted on the basis of the [prime / alternate] key.

prime

6 The ALTERNATE KEY associated with a record [must / need not] be a unique key value in the file.

need not

PROCEDURE DIVISION

An indexed file can be opened as INPUT, OUTPUT, or I/O (input-output). Figure 13-5 summarizes the permissible input-output statements for each of these options, depending on the access mode specified.

Reading records from an indexed file is done by using one of the two formats presented in Figure 13-6.

The first format must be used if the SEQUENTIAL access mode has been specified either explicitly or implicitly (by default). The NEXT phrase must be specified when a file is declared to be in the DYNAMIC access mode and records are to be retrieved sequentially. Execution of READ . . . NEXT RECORD retrieves from the file the next record whose record key is higher than the one accessed previously. From a logical standpoint, READ . . . NEXT operates identically to READ in a sequential magnetic tape file. We use the qualification "from a logical standpoint" because in an indexed file the physical and logical order may not be in direct correspondence. This happens when an indexed file has new records added to it. Instead of being "squeezed" in between existing records, they are put in a physically separate location (overflow) and connected by address pointers to the records that logically precede and follow them. The AT END clause has the same meaning as for a sequential file. Similarly, the NOT AT END and the END-READ options associated with the revised version of COBOL have the same meaning as for a sequential file.

Format 2 is used for files in RANDOM access mode; it also is used when records are to be retrieved randomly from a file in DYNAMIC access mode. The KEY clause references the data-name specified as a key either in the RECORD KEY or the ALTERNATE RECORD KEY clauses. If the KEY clause is omitted

FILE ACCESS MODE	STATEMENT	OPEN MODE INPUT	OUTPUT	I/O
Sequential	READ	X		X
	WRITE		X	
	REWRITE			X
	START	X		X
	DELETE			X
Random	READ	X		X
	WRITE		X	X
	REWRITE			X
	START			
	DELETE			X
Dynamic	READ	X		X
	WRITE		X	X
	REWRITE			X
	START	X		X
	DELETE			X

FIGURE 13-5
PERMISSIBLE INPUT-OUTPUT STATEMENTS

in the Format 2 READ statement, the prime key (RECORD KEY) of the file is assumed by default. The INVALID KEY condition holds when no record can be located whose record key matches the value of the data-name specified or implied by the KEY IS clause. For instance, we may have:

```
MOVE '123456789' TO SOC-SEC-NO.
READ STUDENT-MASTER RECORD
          KEY IS SOC-SEC-NO
          INVALID KEY PERFORM READ-ERROR
          NOT INVALID KEY PERFORM ROUTINE-A
                              PERFORM ROUTINE-B
END-READ
PERFORM ROUTINE-C.
```

In the above program segment, READ-ERROR will be executed if no record in the file has a SOC-SEC-NO key equal to 123456789. If such a record does exist, then ROUTINE-A and ROUTINE-B are executed. Notice that ROUTINE-C is executed regardless of the outcome of the READ because of the END-

```
Format 1:
    READ file-name-1   [NEXT] RECORD [INTO identifier-1]
        [AT END imperative-statement-1]
        [NOT AT END imperative-statement-2]
    [END-READ]
Format 2:
    READ file-name-1 RECORD [INTO identifier-1]
        [KEY IS data-name-1]
        [INVALID KEY imperative-statement-3]
        [NOT INVALID KEY imperative-statement-4]
    [END-READ]
```

FIGURE 13-6
GENERAL FORMATS FOR THE READ STATEMENT FOR INDEXED FILES.

READ scope-terminator. Also, SOC-SEC-NO is of course a key field. Therefore, one of the following two clauses is assumed to have been written in the SELECT statement:

```
RECORD KEY IS SOC-SEC-NO
or
ALTERNATE RECORD KEY IS SOC-SEC-NO.
```

It is instructive also to consider rewriting the above example using the FILE STATUS construct. Assume that in the SELECT statement, we had written the clause:

```
FILE STATUS IS STUD-MAST-STATUS.
```

Then the example could be rewritten as follows:

```
    MOVE '123456789' TO SOC-SEC-NO
    READ STUDENT-MASTER RECORD
         KEY IS SOC-SEC-NO
         INVALID KEY PERFORM CHECK-STATUS-CODE
         NOT INVALID KEY PERFORM ROUTINE-A
                         PERFORM ROUTINE-B
    END-READ
    PERFORM ROUTINE-C.
CHECK-STATUS-CODE.
    IF STUD-MAST-STATUS = '23'
         PERFORM READ-ERROR
    ELSE
         IF STUD-MAST-STATUS = . . .
```

When a FILE STATUS clause has been specified, and prior to the execution of the INVALID . . . and NOT INVALID . . . tests, the system places the appropriate status code in the FILE STATUS field. Making reference to the codes in Figure 13-4, we see that a code of '00' signifies successful completion. When a read instruction has been successful in the example above, STUD-MAST-STATUS would contain '00' and the NOT INVALID KEY branch would be executed. Within that branch we could 'look' at the contents of STUD-MAST-STATUS, but since it is known to be '00' there is no point in doing so.

When the read operation is not successful, the INVALID KEY clause applies, and we execute CHECK-STATUS-CODE in which we proceed to test for the specific error condition. A code of '23' signifies attempt to read a nonexistent record, as shown in Figure 13-4. Other status codes could also be tested in order to ascertain the specific cause of the INVALID KEY condition.

As a summary statement, the INVALID KEY condition is generic, whereas the FILE STATUS construct allows us to identify the specific invalid condition.

We take the opportunity here to suggest a more documentative presentation of the file-status specification. Assuming, as above, that we had written FILE STATUS IS STUD-MAST-STATUS, we could write:

```
01 STUD-MAST-STATUS   PIC   XX.
   88 VALID-READ   VALUE '00'.
   88 NONEXISTENT-RECORD VALUE '23.'
   88 (etc.)
   .
   .
   .
```

Such a specification would allow replacing statements like IF STUD-MAST-STATUS = '23' by IF NONEXISTENT-RECORD, and would remove the hard-to-remember numeric codes from the PROCEDURE DIVISION.

```
WRITE record-name-1 [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
[END-WRITE]

REWRITE record-name-1 [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
[END-REWRITE]

DELETE file-name-1 RECORD
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
[END-DELETE]
```

FIGURE 13-7

GENERAL FORMATS FOR WRITE, REWRITE, AND DELETE STATEMENTS FOR INDEXED FILES.

New records are written in the file by use of the WRITE statement presented in the first portion of Figure 13-7. When a file is being created, WRITE is used as illustrated in the example included in the preceding section of this chapter.

After a file has been created, WRITE is used to add new records to the file. As always, the proper value is moved to the primary record key, and the execution of WRITE causes the new record to be inserted in the correct logical position within the file. The INVALID KEY condition is true under the following circumstances: (1) when the file has been opened as OUTPUT, and the value of the primary record key is not greater than the value of the primary record key of the previous record; (2) when the file has been opened as I/O, and the value of the primary record key is equal to the value of the primary record key of a record already existing in the file; (3) when an attempt is being made to write more records than can be accommodated by the available disk storage.

As with the READ verb, the WRITE allows for NOT INVALID . . . END-WRITE options. Further, both the INVALID . . . and NOT INVALID clauses are optional and can be replaced by corresponding FILE STATUS related statements, as illustrated in the preceding program segment for the READ statement.

In updating tasks, REWRITE is used to replace a record that exists in the file. The general format is included in the middle portion of Figure 13-7.

At the time of execution of REWRITE, the file must be open in the I/O mode. The record being replaced is the one whose key matches the value of the primary record key. The INVALID KEY holds when the value of the record key in the record to be replaced does not match the value of the record key of the last record read, or the value of the record key does not equal the record key of any record existing in the file.

The DELETE statement logically removes a record from an indexed file. The general format is shown in the last portion of Figure 13-7.

A DELETE command can be executed only if the file has been opened in the I/O mode because a DELETE statement must have been preceded by a successful READ statement. Notice the explanations associated with codes 43 and 49 in Figure 13-4, which are file-status codes for invalid DELETE operations.

If the file has been declared to be in RANDOM or DYNAMIC access mode, INVALID KEY is true when the file does not contain a record whose prime record key value matches the value of the record key. Thus, the programmer is responsible for moving the key value of the record to be deleted to RECORD KEY.

```
                          ┌ IS EQUAL to                          ┐
                          │ IS =                                 │
                          │ IS GREATER THAN                      │
START file-name-1  [KEY  <  IS >                                   > data-name-1 ]
                          │ IS NOT LESS THAN                     │
                          │ IS NOT <                             │
                          │ IS GREATER THAN OR EQUAL TO          │
                          └ IS > =                               ┘
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
  [END-START]
```

FIGURE 13-8
GENERAL FORMAT FOR THE START STATEMENT FOR INDEXED FILE.

The START verb allows sequential retrieval of records from a point other than the beginning of the file. Thus, it is possible to retrieve records sequentially starting with some record in the "middle" of the file, as shown in the general format for the START statement in Figure 13-8.

The file must be in SEQUENTIAL or DYNAMIC access mode and must be open in the INPUT or I/O mode at the time START is executed. The KEY phrase may be omitted, in which case EQUAL is implied. In essence, the START statement means to position the file to that record whose record key satisfies the explicit or implicit KEY condition. If we simply write:

```
START CUSTOMER FILE   INVALID KEY PERFORM CANT-START
```

we specify that the file is to be positioned at the record whose primary key has a value equal to the current content of the RECORD KEY field. Thus, it would be important to have ascertained that the RECORD KEY has an appropriate value. For instance if the RECORD KEY field contained the value of a deleted record, we would execute CANT-START, since the INVALID KEY condition would be true.

A more typical example would involve use of the KEY clause to specify the key name (date-name-1), and the condition for determining the "match." For example, if CUSTOMER-NAME is a record key and we want to retrieve the records of customers whose names begin with M or higher, we can write:

```
MOVE 'M' TO CUSTOMER-NAME
START CUSTOMER-FILE
      KEY IS NOT LESS THAN CUSTOMER-NAME
      INVALID KEY PERFORM START-ERROR.
```

In this program segment, MOVE is an alphanumeric move resulting in CUSTOMER-NAME containing the letter M and blanks to the right of M. The KEY IS NOT LESS condition specifies that we want to position the file at the first record whose key is not less than the letter M; in other words, it will be the first name that begins with the letter M. A READ statement then retrieves that record, and subsequent READ statement execution retrieves the following records sequentially.

The data-name in the KEY phrase can be either a record key (specified as RECORD KEY, or ALTERNATE RECORD KEY) or a data item subordinate to a record key, provided that the data item is the first (leftmost) field in the record key. In other words we can specify the "first part" of a record key. For the above example, we could have:

```
    . . . RECORD KEY IS CUSTOMER-NAME . . .
    .
    .
    .
02  CUSTOMER-NAME.
    03  STARTING-LETTER-OF-NAME       PIC  X.
    03  REST-OF-NAME                  PIC  X(19).
    .
    .
    .
    MOVE 'M' TO STARTING-LETTER-OF-NAME
    START CUSTOMER-FILE
        KEY IS GREATER THAN OR EQUAL TO STARTING-LETTER-OF-NAME
            INVALID KEY PERFORM START-ERROR.
```

The INVALID condition is true if the KEY condition cannot be met. For example, consider the following program segment:

```
MOVE 'MICHENER' TO CUSTOMER-NAME.
START CUSTOMER-FILE
    KEY IS EQUAL TO CUSTOMER-NAME
    INVALID KEY PERFORM ERROR-START.
```

ERROR-START will be executed if there is no customer in the file whose record key is equal to 'MICHENER'.

As with all I/O verbs (OPEN, CLOSE, READ, WRITE, REWRITE, DELETE, and START) the FILE STATUS may be used as a more discriminating alternative to the generic INVALID KEY . . . NOT INVALID KEY . . . END-verb approach.

Review

1 In the PROCEDURE DIVISION, reading records from an indexed file is accomplished by using the ________________ verb.

READ

2 Whenever records in an indexed file are to be retrieved randomly, the ________________ clause indicates the data-name to be used to identify each record.

KEY

3 If the KEY clause is omitted in conjunction with random retrieval of records, the basis for identifying each record is the [RECORD KEY / ALTERNATE RECORD KEY].

RECORD KEY

4 In the PROCEDURE DIVISION, records are added to an indexed file by using the ________________ verb.

WRITE

5 Execution of a WRITE statement to add a record to an indexed file results in the record being added [at the end of the file / in the correct logical position within the file].

in the correct logical position within the file

6 The verb that is used to modify a record held in an indexed file is __________________.

REWRITE

7 In order for a REWRITE instruction to be executed, the file must be open in __________________ mode.

I/O

8 The verb used to remove a record from an indexed file is __________________.

DELETE

9 A DELETE command can be executed only if the file has been opened in __________________ mode.

I/O

10 The verb that makes possible the sequential retrieval of records from a point other than the beginning of the file is __________________.

START

11 In conjunction with executing a START statement, the data-name used as the key [must / need not] have been specified previously as RECORD·KEY or ALTERNATE RECORD KEY.

need not

AN EXAMPLE OF PROCESSING AN INDEXED FILE

We present an example here that illustrates the use of most of the language statements and options described in the preceding section. The example involves processing the vendor file created by the example program in Figure 13-2. We now give the following record description to the file.

```
FD  VENDOR-FILE LABEL RECORDS ARE STANDARD
                DATA RECORD IS DISK-RECORD.
01  VENDOR-RECORD.
    02  VENDOR-NUMBER       PIC X(8).
    02  VENDOR-NAME         PIC X(15).
    02  VENDOR-ADDRESS      PIC X(45).
```

Transaction records are submitted through a TRANS-FILE and have the following record description:

```
01  TRANS-RECORD.
    02  TRANS-CODE                              PIC 9.
        88  CHANGE-ADDRESS                      VALUE 1.
        88  ADD-VENDOR                          VALUE 2.
        88  DELETE-VENDOR                       VALUE 3.
        88  ERROR-CODE VALUES ARE ZERO, 4 THRU 9.
    02  VENDOR-IDENT                            PIC 9(8).
    02  VENDOR-NAME                             PIC X(15).
    02  VENDOR-ADDRESS                          PIC X(45).
    02  FILLER                                  PIC X(11)
```

It is apparent from the self-documenting nature of this record description that we are interested in changing the address of a vendor, and in adding or deleting vendors.

Figure 13-9 presents the complete program. Notice in the ENVIRONMENT

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. UPDATEINDEX.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT VENDOR-FILE ASSIGN TO OLDMSTR
                        ORGANIZATION IS INDEXED
                        ACCESS MODE IS DYNAMIC
                        RECORD KEY IS VENDOR-NUMBER.
*
     SELECT TRANS-FILE ASSIGN TO TRANS.
*
     SELECT PRINT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
*
 FD  VENDOR-FILE LABEL RECORDS ARE STANDARD
                 DATA RECORD IS VENDOR-RECORD.
 01  VENDOR-RECORD.
     02 VENDOR-NUMBER      PIC 9(8).
     02 VENDOR-NAME        PIC X(15).
     02 VENDOR-ADDRESS     PIC X(45).
*
 FD  TRANS-FILE LABEL RECORDS ARE OMITTED
               DATA RECORD IS TRANS-RECORD.
 01  TRANS-RECORD.
     02 TRANS-CODE         PIC 9.
        88 CHANGE-ADDRESS  VALUE 1.
        88 ADD-VENDOR      VALUE 2.
        88 DELETE-VENDOR   VALUE 3.
        88 ERROR-CODE      VALUES ARE ZERO, 4 THRU 9.
     02 VENDOR-IDENT       PIC 9(8).
     02 VENDOR-NAME        PIC X(15).
     02 VENDOR-ADDRESS     PIC X(45).
     02 FILLER             PIC X(11).
*
 FD  PRINT-FILE LABEL RECORD OMITTED
               DATA RECORD PRINTLINE.
 01  PRINTLINE          PIC X(132).
*
 WORKING-STORAGE SECTION.
 01  END-OF-DATA-INDICATOR   PIC 9 VALUE ZERO.
     88 INPUT-ENDED        VALUE 1.
/
 PROCEDURE DIVISION.
 MAIN-ROUTINE.
*
*  THIS PORTION ILLUSTATES RANDOM UPDATING OF INDEXED FILE
*
     OPEN INPUT TRANS-FILE
         OUTPUT PRINT-FILE
            I-O   VENDOR-FILE.
     MOVE ' LISTING FROM UPDATE PORTION' TO PRINTLINE
     WRITE PRINTLINE AFTER ADVANCING PAGE.
     PERFORM READ-TRANS.
     PERFORM UPDATE UNTIL INPUT-ENDED.
     CLOSE VENDOR-FILE.
*
*  THIS PORTION ILLUSTRATES SEQUENTIAL RETRIEVAL.
*
     MOVE ZERO TO END-OF-DATA-INDICATOR.
     MOVE ' LISTING FROM SEQUENTIAL RETRIEVAL' TO PRINTLINE
     WRITE PRINTLINE AFTER ADVANCING 5 LINES.
     OPEN INPUT VENDOR-FILE
     PERFORM READ-VENDOR.
```

FIGURE 13-9

SAMPLE PROGRAM TO ILLUSTRATE PROCESSING OF AN INDEXED FILE.

```
      *
           PERFORM LISTING UNTIL INPUT-ENDED.
           CLOSE VENDOR-FILE.
      *
      *  THIS PORTION ILLUSTRATES USE OF THE START VERB.
      *
           MOVE ZERO TO END-OF-DATA-INDICATOR
           MOVE 35290001 TO VENDOR-NUMBER
           OPEN INPUT VENDOR-FILE
      *
           START VENDOR-FILE KEY IS GREATER THAN VENDOR-NUMBER
               INVALID KEY MOVE 1 TO END-OF-DATA-INDICATOR.
      *
           MOVE ' LISTING FROM USE OF START VERB' TO PRINTLINE.
           WRITE PRINTLINE AFTER ADVANCING 5 LINES.
           PERFORM READ-VENDOR
           PERFORM LISTING UNTIL INPUT-ENDED
           CLOSE VENDOR-FILE.
      *
           CLOSE TRANS-FILE.
           STOP RUN.
      *
       READ-TRANS.
           READ TRANS-FILE RECORD
               AT END MOVE 1 TO END-OF-DATA-INDICATOR.
       UPDATE.
           MOVE VENDOR-IDENT OF TRANS-RECORD TO VENDOR-NUMBER.
           IF CHANGE-ADDRESS
              PERFORM ADDRESS-1 THRU ADDRESS-3
           ELSE
              IF ADD-VENDOR
                 PERFORM ADDITION-1 THRU ADDITION-3
              ELSE
                 IF DELETE-VENDOR
                    PERFORM DELETION-1 THRU DELETION-3
                 ELSE
                    PERFORM ERROR-TRANS.
           PERFORM READ-TRANS.
      *
       ADDRESS-1.
           READ VENDOR-FILE RECORD
               INVALID KEY GO TO ADDRESS-2-0.
           MOVE VENDOR-ADDRESS OF TRANS-RECORD
               TO VENDOR-ADDRESS OF VENDOR-RECORD.
           REWRITE VENDOR-RECORD
               INVALID KEY GO TO ADDRESS-2-1.
           GO TO ADDRESS-3.
       ADDRESS-2-0.
           MOVE ' CANNOT FIND VENDOR FOR THIS TRANS ' TO PRINTLINE
           WRITE PRINTLINE AFTER ADVANCING 1 LINE.
           WRITE PRINTLINE FROM TRANS-RECORD AFTER ADVANCING 1 LINE.
           GO TO ADDRESS-3.
       ADDRESS-2-1.
           MOVE ' CANNOT REWRITE THIS RECORD' TO PRINTLINE
           WRITE PRINTLINE AFTER ADVANCING 1 LINE
           WRITE PRINTLINE FROM VENDOR-RECORD AFTER 1.
           GO TO ADDRESS-3.
       ADDRESS-3.
           EXIT.
      *
       ADDITION-1.
           MOVE VENDOR-NAME OF TRANS-RECORD
               TO VENDOR-NAME OF VENDOR-RECORD
           MOVE VENDOR-ADDRESS OF TRANS-RECORD
               TO VENDOR-ADDRESS OF VENDOR-RECORD.
           WRITE VENDOR-RECORD
               INVALID KEY GO TO ADDITION-2.
           GO TO ADDITION-3.
       ADDITION-2.
           MOVE ' CANNOT CREATE A RECORD FROM THIS TRANS' TO PRINTLINE
           WRITE PRINTLINE AFTER 1
```

FIGURE 13-9 *(Continued)*

SAMPLE PROGRAM TO ILLUSTRATE PROCESSING OF AN INDEXED FILE.

```
     WRITE PRINTLINE FROM TRANS-RECORD AFTER 1.
     GO TO ADDITION-3.
 ADDITION-3.
     EXIT.
 DELETION-1.
     DELETE VENDOR-FILE RECORD
            INVALID KEY GO TO DELETION-2.
     GO TO DELETION-3.
 DELETION-2.
     MOVE ' CANNOT DELETE RECORD SPECIFIED BY THIS TRANS'
              TO PRINTLINE
     WRITE PRINTLINE AFTER ADVANCING 1 LINE
     WRITE PRINTLINE FROM TRANS-RECORD AFTER 1.
     GO TO DELETION-3.
 DELETION-3.
     EXIT.
*
 ERROR-TRANS.
     MOVE ' WRONG TRANSACTION CODE IN ' TO PRINTLINE
     WRITE PRINTLINE AFTER ADVANCING 1 LINE
     WRITE PRINTLINE FROM TRANS-RECORD AFTER 1.
*
 READ-VENDOR.
     READ VENDOR-FILE NEXT RECORD
          AT END MOVE 1 TO END-OF-DATA-INDICATOR.
*
 LISTING.
     WRITE PRINTLINE FROM VENDOR-RECORD AFTER 1.
     PERFORM READ-VENDOR.
```

FIGURE 13-9 *(Continued)*
SAMPLE PROGRAM TO ILLUSTRATE PROCESSING OF AN INDEXED FILE.

DIVISION that VENDOR-FILE is in DYNAMIC access mode and that VENDOR-NUMBER is the record key. The PROCEDURE DIVISION is self-documenting and consists of three control portions that illustrate random updating, sequential retrieval, and use of the START verb, respectively. The first portion illustrates random access and updating. The following sample input records were used:

211111111ACME-01	ADDED RECORD
135120001	NEW ADDRESS
497654310	ERROR CODE TEST = 4
349678912	DELETE
160000000	NEW ADDRESS
349678912	DELETE PREVIOUSLY DELETED
169743210	CHANGE NONEXISTING

We have written comments instead of data to simplify debugging. For instance, the first record has a code of 2 (column 1), which means to add a new vendor whose number is 11111111. Instead of address, we have written the comment ADDED RECORD.

The first portion of the MAIN-ROUTINE paragraph in the PROCEDURE DIVISION illustrates random updating of an indexed sequential file. By studying Figure 13-9, we can see that the UPDATE paragraph analyzes the transaction code. Then we execute the ADDRESS, ADDITION, DELETION, or ERROR-TRANS paragraph. Each of these paragraphs illustrates, respectively: the replacing of a record (REWRITE), the addition of a record (WRITE), the deletion of a record (DELETE), and the handling of INVALID KEY conditions.

The second portion of the MAIN-ROUTINE procedure illustrates sequential retrieval. When OPEN INPUT VENDOR-FILE is executed, the open instruction causes the file to be positioned at the beginning, so that when the first READ VENDOR-FILE NEXT RECORD is executed, the first record is retrieved.

```
LISTING FROM UPDATE PORTION

WRONG TRANSACTION CODE IN
497654310          ERROR CODE TEST = 4
CANNOT DELETE RECORD SPECIFIED BY THIS TRANS
349678912          DELETE PREVIOUSLY DELETED
CANNOT FIND VENDOR FOR THIS TRANS
169743210          CHANGE NONEXISTENT

LISTING FROM SEQUENTIAL RETRIEVAL

11111111ACME-01    ADDED RECORD
12345678ACME-10
22345678ACME-20
35120001ACME-30    NEW ADDRESS
37890123ACME-40
43215678ACME-50
50000000ACME-60
60000000ACME-70    NEW ADDRESS
61234567ACME-80
70000000ACME-88
88888888ACME-89
90000000ACME-90
99999999ACME-99

LISTING FROM USE OF START VERB

37890123ACME-40
43215678ACME-50
50000000ACME-60
60000000ACME-70    NEW ADDRESS
61234567ACME-80
70000000ACME-88
88888888ACME-89
90000000ACME-90
99999999ACME-99
```

FIGURE 13-10
SAMPLE OUTPUT FROM THE PROGRAM IN FIGURE 13-9.

The third portion of the MAIN-ROUTINE illustrates use of the START verb:

```
MOVE 35290001 TO VENDOR-NUMBER
        .
        .
START VENDOR-FILE KEY IS GREATER THAN VENDOR-NUMBER
    INVALID KEY MOVE 1 TO END-OF-DATA-INDICATOR.
```

In this illustration, we want to retrieve sequentially all records whose key is greater than 35290001. The record whose key equals 35290001 will not be retrieved, since the GREATER THAN option is used.

Figure 13-10 presents the output resulting from executing the program in Figure 13-9. The reader will find it useful to run the programs in Figures 13-2 and 13-9 and obtain the results shown in Figure 13-10. Since compilers differ, some changes to "localize" the programs may be necessary, and the task may prove well worth the effort. Of course, the small volume of data involved allows easy desk checking of the program.

EXERCISES

13.1 Outline a program to re-create an indexed sequential file. Assume that the original file has too many overflow records and that we therefore want to create a new version of the file to eliminate all overflow. Include in your outline the FILE-CONTROL paragraph of the ENVIRONMENT DIVI-

SION and the complete PROCEDURE DIVISION. Assume that OLD-FILE, OLD-REC, NEW-FILE, and NEW-REC are the corresponding file- and record-names.

13.2 Refer to Exercise 9.4 for the program description.

- **a** Create the master file as an indexed file, using the part number as the RECORD KEY.
- **b** Update the master file on a random basis, using the part number as the RECORD KEY.

13.3 Refer to Exercise 9.5 for the program description.

- **a** Create the master file as an indexed sequential file, using the lot number as the RECORD KEY.
- **b** Update the master file on a random basis, using the lot number as the RECORD KEY.

14 RELATIVE FILE PROCESSING

RELATIVE FILE ORGANIZATION

A relative file is one in which records are accessed by reference to their relative position in the file. If we think of a file that can hold 100 records, the first record has a *relative key* of 1, while the last one has a relative key of 100. Access to records in a file organized as relative is by reference to the relative key of each record. For instance, we may use the following two types of commands:

1 Write this record as the 20th record in the file.
2 Read the 68th record in the file.

As this shows, reference is made to the relative location of a record. We say "relative" as distinguished from the *absolute* location of a given record. An absolute location would be specified in terms of the address of the record within a specific disk volume, within a specific cylinder, within a specific track, and the record number in that track.

Relative file organization is ideal for a case where records are identified by consecutive numbers. For instance, suppose that invoice records are numbered 0001, 0002, 0003 . . . We can use relative organization and store the records in the order of the invoice number. If we want to access invoice number 0050, we can do so simply by accessing the record whose relative key is 50. It is rare, however, that records can be numbered exactly consecutively as in this example, and relative file organization requires some specific techniques in order to utilize this method of file organization.

There are several key-to-address transformation methods that can be used to transform record keys (identifiers) to relative location addresses. All such methods have one common property: they enable us to transform record identifiers, such as social security numbers, product numbers, and customer names, to relative

key values. Thus, if we want the record of product number 1234, we would apply a key-to-address transformation procedure that would transform product number 1234 into a relative key value, say 371. Then we would retrieve the 371st record in the relative file, with the expectation that it would be the record of product number 1234.

As the preceding example illustrates, a relative file can be a high-performance file. We can compute the address of a record without having to use index structures, as is required with indexed sequential files. It generally is true that relative files can provide fast, direct access to records, but it should be added that the level of performance depends on the specific file and the specific method chosen. Good performance in a relative file often requires some analysis and experimentation. The main reason for variability in performance is that all known key-to-address methods produce *synonyms*, which are said to occur when the key-to-address method generates the same relative key for two or more data records. For instance, it may be that product numbers 1234 and 4965 both generate the same relative key; therefore, we need additional processing beyond the key-to-address methods.

There are many ways to handle the occurrence of synonyms, but we illustrate here one procedure that is successful. At file creation time, go through the file and mark each record as a vacant record, using a special field in the record for this purpose. Then, as each data record is being stored in the file, we apply the chosen key-to-address method and compute a *home-address* relative key value. We now read that record space, and, if it is marked as vacant, we record the data in that space and then mark it as occupied. If we previously stored a data record in the home-address space, we look at the next record space to see if it is vacant and, if it is, we write the data record there. If it is occupied, we continue looking for a vacancy in consecutive record spaces, and either we find a vacancy or we come full circle in the case of a file that is completely filled. For example, suppose that V = vacant, and O = occupied. Let us assume that we want to store a product record for product number 2645 in the file. Further, let us assume that the chosen key-to-address method computes a relative key of 132. As can be observed in the following, the product record will be stored in relative record location 134, which is the first available vacancy.

RECORD RELATIVE KEY	130	131	132	133	134	135
Content	V	O 7391	O 0935	O 5383	V	V

If we wanted to retrieve the record of product number 2645 after it had been loaded into the file, we would compute its home address as 132 as before. We would attempt to retrieve the record at location 132 by comparing the record key of 2645 to the record key at location 132, which happens to be 0935. This comparison indicates that record 2645 is a synonym, and we would continue searching sequentially until the record was found at location 134. Notice that, if we were trying to retrieve a record that in fact did not exist in the file, we would know that the record did not exist as soon as we came upon the first vacancy. We could conclude that the record did not exist because if a record is a synonym it is stored in the *first* available vacancy past the home address.

This procedure needs to be expanded to take account of deletion of records from the file. If we desire to free up the record space when a record is deleted, we recommend that the space be marked as deleted. If we want to add a new record, a deleted record then constitutes a vacancy. If we are retrieving a record and it is not at its home address, however, we should continue looking for it until we encounter the first vacancy, *not* counting deleted records as vacancies. In the

preceding example of record 2645, whose home address was 132 but which was written at 134 due to synonyms, suppose that the record at location 133 was later deleted. To retrieve record 2645 from location 134, we would have to continue looking past the deleted record at location 133. Of course, if the record at location 133 were deleted *before* record 2645 was added to the file, then record 2645 would have been written at location 133.

This procedure for handling synonyms is incorporated into the program illustration in Figure 14-7, later in this chapter.

Review

1 In a relative file, the location of a record is described relative to the other ________ in the file.

records

2 Any key-to-address transformation method is concerned with converting a record key (identifier) into a relative ________________.

location address

3 When two different record keys result in the same relative location address, it is said that a ________________ has been produced.

synonym

4 When a synonym has occurred, a common solution is to store the synonym in the first vacant record space [preceding / following] the home-address space.

following

5 When records are deleted from a relative file and are marked as deleted, such record spaces then [are / are not] available for the storage of new records.

are

6 If a record is neither at its home address nor at the first vacancy after that address (not counting deleted records), then we can conclude that ________________________________.

the record does not exist

THE DIVISION REMAINDER METHOD

In the preceding section, we saw that use of relative file organization is predicated on a key-to-address transformation capability. There are several key-to-address methods available. One that is in wide use is the division remainder method.

In order to apply the division remainder method, we first must choose the file size. A rule of thumb to determine the file size is to divide the number of records in the file by 0.80, so that 80 percent of the file will be filled and 20 percent of the file will be unused.* This unused space serves two purposes: it cuts down the incidence of synonyms and it also allows for file expansion. The term *packing*

*Where X = file size
If No. Records = .80X

$$\text{Then } X = \frac{\text{No. Records}}{.80}$$

factor or *file density* is used to denote the percent of filled space in a relative file. If we divide the number of data records by 0.80, the result is an 80 percent packing factor, or density. In general, the lower the packing factor the fewer the synonyms but the greater the unused file space. We can see now why relative files require analysis and experimentation. For instance, record size and frequency of access are factors that would be considered in the analysis for choosing a suitable packing factor.

Once the file size has been chosen, we select a prime number closest to the file size. A prime number is divisible only by itself and the number 1. For instance, the numbers 3 and 7 are prime numbers. There are tables of prime numbers available in mathematical handbooks. Use of a prime number is not critical, however. We also can use the file size instead of the prime number closest to it, and in most cases there is little difference in the number of synonyms generated. In general, however, the use of a prime number is likely to result in fewer synonyms.

Suppose a customer file contains 9,600 records. We decide on a packing factor of 80 percent; therefore, the allocated file size is 9,600 ÷ 0.80 = 12,000 record spaces. The prime number closest to 12,000 is 11,987. The customers are identified by a six-digit customer number. The first step in the division remainder method is to divide the record identifier by the prime number (or the file size itself). Thus, if we want to compute the location address of customer 123456, we divide this record identifier by the prime number 11987, obtaining a whole-number quotient of 10 and a remainder of 3586. By the division remainder method, the remainder plus 1 is the computed home address. For our example, the address is thus 3586 + 1 = 3587. If we had used the file size instead of the prime number as the divisor, we would have divided 123456 by 12000, giving a whole-number quotient of 10 and a remainder of 3456. The home address in this case would have been 3456 + 1 = 3457.

It should be noted that, when we divide a number by any divisor, the remainder from the division can range from zero to the value of the divisor minus one. Thus, if we are dividing by 12000, the smallest possible remainder is zero and the largest possible remainder is 11999. Since we add one to the remainder in the division remainder method, we can see that division by the file size results in the smallest computed value being one and the largest value being the file size itself. Thus, by taking the remainder plus one, we generate addresses that correspond to the relative addresses in the file. In the case where a prime number is used that is smaller than the file size itself, a few addresses would be impossible to generate. For example, with a prime number of 11987 as a divisor and a file size of 12000, we would never generate addresses 11988, 11989, 11990, . . . 12000. Still, those addresses could be occupied by records if we happened to have enough synonyms in the addresses 11987 and before.

As indicated by this example, the division remainder method is rather easy to apply. In the same program given later in the chapter (Figure 14-7), the procedure is implemented as follows:

```
MOVE ITEM-NUMBER OF SOURCE-RECORD TO WORKFIELD
DIVIDE PRIME-NUMBER INTO WORKFIELD GIVING QUOTIENT
COMPUTE LOCATION-ADDRESS =
        WORKFIELD - (PRIME-NUMBER * QUOTIENT) + 1.
```

The QUOTIENT field must be defined as an integer field, of course. Then the remainder is computed by multiplying the divisor by the integer quotient and then subtracting the result from the dividend. The address is stored in LOCATION-ADDRESS as the remainder plus one.

The division remainder method, as well as other methods for transforming

record identifiers into file addresses, has the property of giving about an equal chance to every possible file address. The general relationship between the record identifier and the address computed is random, and this is the reason that key-to-address methods are referred to as *randomizing* methods. The term *hashing* also is used very widely to describe such randomizing key-to-address methods.

Review

1 A key-to-address transformation is required for relative files in order to convert a record key (identifier) into a relative file address. The widely used method described in this section is called the ________________ method.

division remainder

2 The percentage of filled space in a relative file is identified by the term ________________.

packing factor (or file density)

3 Suppose an accounts payable file includes 2,100 records. If a 70 percent packing factor is to be used, the allocated file size would be ________ record spaces.

2,100 ÷ 0.70 = 3,000

4 Given a file size of 3,000, transform Account No. 4211 into a file address by using the file size as the divisor. ________

Integer quotient: 1
Remainder: 1211
File Address: 1211 + 1 = 1212

5 Given a file size of 3,000, transform Account No. 911 into a file address by using the file size as the divisor. ________

Integer quotient: 0
Remainder: 911
File Address: 911 + 1 = 912

6 Key-to-address methods are called randomizing methods because the possible [record identifiers/file addresses] have about an equal chance of occurring.

file addresses

OTHER KEY-TO-ADDRESS TRANSFORMATION METHODS

Digit Analysis Method

By this method a frequency count is performed in regard to the number of times each of the 10 digits occurs in each of the positions included in the record key. For example, Table 14-1 presents a frequency count for the number of times each digit occurred in a five-position numeric key for 2,800 records. In this tabulation we can observe that digits 0–9 occur with approximately uniform distribution in key positions 2, 3, and 5; therefore, if a three-digit address were required, the digits in these three positions in the record keys could be used. Given that there are 2,800 records, however, a four-digit address would be required. Suppose we desire the first digit to be a 0, 1, 2, or 3 only. Such assignment can be made with about equal frequency for each digit by using a rule such as the following: assign a "0" when digits in positions 2 and 3 both contain odd numbers, a "1" if position 2 is odd and position 3 is even, a "2" if position 2 is even and position 3 is odd, or a "3" if positions 2 and 3 both contain

TABLE 14-1
FREQUENCY OF OCCURRENCE OF THE DIGITS 169 FOR 2,800 FIVE-POSITION KEYS

	KEY POSITION				
DIGIT	1	2	3	4	5
0	2026				
1	618	250	218	1012	260
2	128	395	391	185	382
3	23	263	389	299	271
4	5	298	330	52	302
5		335	299	101	387
6		303	339	18	299
7		289	308	134	301
8		267	267	999	245
9		400	259		353

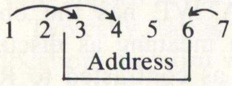
even numbers. Thus, the address for key 16258 would be 3628: the "3" from the fact that positions 2 and 3 both contain even numbers and the "628" from key positions 2, 3, and 5. Other rules for prefixing additional digits can be formulated for different circumstances. In any event, the digit analysis method relies on the digits in some of the key positions being approximately equally distributed. If such is not the case, the method canot be used with good results.

Mid-Square Method

The record key is multiplied by itself, and the product is truncated from both left and right so as to form a number equal to the desired address length. Thus, key 36258 would be squared to give 1314642564. To form a four-digit address, this number would be truncated from both the left and right, resulting in the address 4642.

Folding

The key is separated into two parts which then are added together to form the address. For example, suppose key 1234567 is to be transformed into a four-digit address. We can add the first four positions to the last three positions to form the address; in this case: 1234 + 567 = 1801. As another possibility, we can begin with the middle four digits and add the other digits as follows:



In general, the concept of folding does not refer to one standard method but to a general class of possibilities.

Alphabetic Keys

It is possible and sometimes common that the key is alphabetic, as in the case of a student file that utilizes an alphabetic key. In order to determine a numeric address, a procedure is defined by which letters are transformed into numbers. These numbers then might be used as addresses or, more likely, might be used in conjunction with one of the randomizing techniques discussed previously. Thus, if the transformation rule is that A = 00, B = 01, . . . Z = 25, then ADAM would become 00030012.

The key-to-address transformation methods that have been discussed are not the only ones that can be used, but they do represent the principal techniques. As indicated previously, the division remainder technique is used most frequently and generally works at least as well as other methods, but special circumstances may make some other method desirable for a given file.

Review

1 The transformation method for which the digits in at least some of the key positions must be dispersed about equally in terms of value is the ________ method.

digit analysis

2 The transformation method in which the key is multiplied by itself as part of the procedure for determining the address for the record is the ________ method.

mid-square

3 The transformation method in which one part of a key number is added to another part of the number to form the address is the ________ method.

folding

4 Alphabetic keys generally are transformed [directly into a numeric address/ into a numeric code for subsequent determination of an address].

into a numeric code for subsequent determination of an address

5 The transformation technique most frequently used in conjunction with relative file organization is the ________ method.

division remainder

COBOL STATEMENTS FOR RELATIVE FILES

The SELECT Statement

Matters such as key-to-address transformations and handling of synonyms are not acknowledged by the language. COBOL assumes that the programmer handles these. The language provides only the basic mechanism by which relative files can be created and processed.

The general format for the SELECT statement is presented in Figure 14-1. ORGANIZATION IS RELATIVE has the obvious meaning. The ACCESS MODE clause has the same meaning as discussed for indexed files. Notice, however, RELATIVE KEY as contrasted to RECORD KEY. The data-name specified as RELATIVE KEY must be a WORKING-STORAGE unsigned integer item. Its function is to contain the location address for the record about to be accessed, or the location of the record that was just accessed.

```
SELECT file-name

ASSIGN TO {implementor-name}
          {literal         }

[ORGANIZATION IS] RELATIVE

[               {SEQUENTIAL [RELATIVE KEY IS data-name-1]} ]
[ ACCESS MODE IS {RANDOM                                  } ]
[               {DYNAMIC     RELATIVE KEY IS data-name-1  } ]

[FILE STATUS IS data-name-2]
```

FIGURE 14-1
GENERAL FORMAT FOR THE SELECT STATEMENT.

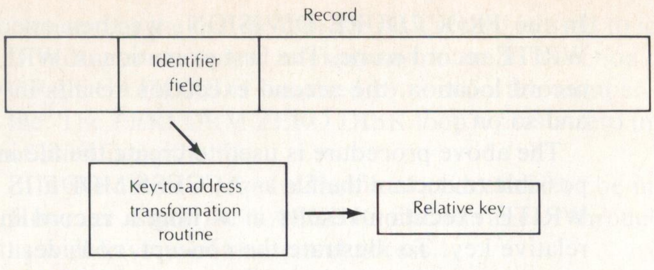


FIGURE 14-2
ILLUSTRATION OF THE ROLE OF RELATIVE KEY.

STATUS CODE	DESCRIPTION
00	Successful execution.
04	Record length does not conform to file description.
10	End of file while attempting a sequential read.
22	Attempt to write a record where a record is already written (perhaps, a REWRITE should have been used).
23	Attempt to read a record that does not exist.
24	Attempt to write beyond the boundaries of the file.
41	Attempt to OPEN a file that is already opened.
42	Attempt to CLOSE a file that is not open.
43	Attempt to DELETE or REWRITE while in the sequential access mode, yet no preceding successful READ has been executed.
47	Attempt to READ or START a file not in the I/O mode.
48	Attempt to WRITE in a file not open in the OUTPUT, I/O, or EXTEND mode.
49	Attempt to DELETE or REWRITE in a file not in the I/O mode.

FIGURE 14-3
SELECTED FILE STATUS CODES AND THEIR MEANING.

The reader should understand clearly the role of RELATIVE KEY, which is different from the RECORD KEY of indexed files. RECORD KEY is part of the file record. RELATIVE KEY is an item apart from the record. When we want to access the 3rd record we move the value 3 to the RELATIVE KEY field and then we issue an I/O verb such as READ, WRITE, REWRITE, or DELETE. The system then uses the RELATIVE KEY value to determine where the I/O verb applies in the file.

Typically, rather than wanting the 3rd or some other record referenced by its location, a record whose key field has some particular value is desired. Given a record, the value of its identifier field is taken by the programmer and transformed through a key-to-address routine to a location address. Figure 14-2 illustrates the process.

The FILE STATUS clause in Figure 14-1 is identical to its counterpart in sequential and indexed files. Figure 14-3 is a list of selected status codes. For a complete list, the reader should consult the appropriate manual for the computer system used.

Creating a Relative Key

A relative file may be created either sequentially or randomly. If it is to be created sequentially, we may omit the RELATIVE KEY clause, an option shown in Figure 14-1. In the format, observe that the RELATIVE KEY clause is required if the access mode is RANDOM or DYNAMIC. To create the file sequentially, we can write:

```
SELECT file-name ASSIGN TO device
       ORGANIZATION IS RELATIVE
       ACCESS MODE IS SEQUENTIAL.
```

In the PROCEDURE DIVISION, we then open the file as OUTPUT and we WRITE record-name. The first execution of WRITE results in writing in the first record location, the second execution results in writing in the second location, and so on.

The above procedure is used to create the file in the sequential mode. It is also possible to declare the file as ACCESS MODE IS RANDOM, in which case each WRITE execution results in writing a record in the location specified by the relative key. To illustrate the concept, consider this example:

```
SELECT CUSTOMER-FILE ASSIGN TO . . .
       ORGANIZATION IS RELATIVE
       ACCESS MODE IS RANDOM
       RELATIVE KEY IS CUST-KEY.

OPEN OUTPUT CUSTOMER-FILE
PERFORM WRITE-SAMPLE VARYING CUST-KEY FROM 1 BY 2
                     UNTIL CUST-KEY > 9.
.
.
WRITE-SAMPLE.
    MOVE SPACES TO CUSTOMER-RECORD
    WRITE CUSTOMER-RECORD.
```

The above example writes (blank) records in record-locations 1, 3, 5, 7, and 9; all other record-locations do not contain data records at this point. If we attempted to read the 4th record, we would be attempting to read a nonexistent record. Also, if subsequent to the above example we accessed all the records sequentially, there would be five records accessed. In other words, the system keeps track of used and vacant record-locations. Still, we recommend that the programmer initialize all the records in the file with a distinguishing code in a special field. This method allows a more convenient control method for sensing vacant and used record-locations. The following example illustrates the procedure:

```
    SELECT DISK-FILE ASSIGN TO . . .
           ORGANIZATION IS RELATIVE
           ACCESS MODE IS RANDOM
           RELATIVE KEY IS LOCATION-ADDRESS.
01  DISK-RECORD.
    02  REC-STATUS-CODE          PIC 9.
        88  VACANT-RECORD        VALUE ZERO.
        88  USED-RECORD          VALUE 1.
    .
    .
    .
    OPEN OUTPUT DISK-FILE
    PERFORM ZERO-DISK
         VARYING I FROM 1 BY 1 UNTIL I > MAX-NO-OF-LOCATIONS.
    .
    .
    .
ZERO-DISK.
    MOVE I TO LOCATION-ADDRESS
    MOVE ZERO TO REC-STATUS-CODE
    WRITE DISK-RECORD
         INVALID KEY PERFORM . . .
```

Notice that the first field in the disk record is used to identify whether the record is vacant or occupied, as explained in the first section of this chapter. MAX-NO-OF-LOCATIONS is assumed to contain the number of record-locations in the file. The PERFORM ZERO-DISK loop writes a zero in all the records as a means of initializing the file.

In the above example, the file was declared to be in RANDOM access mode. However, since the VARYING clause varies I through all values, we could just as well have used sequential access.

I/O Verbs Used with Relative Files

Input/output verbs used with relative files are similar to the ones discussed in the preceding chapter on indexed files. Figure 14-4 presents the two general formats for the READ statement.

Format 1 must be used if records are retrieved in sequential mode. NEXT must be used if DYNAMIC access mode is specified and records are retrieved sequentially.

Format 2 is used when the access mode is RANDOM, or when the access mode is DYNAMIC and records are retrieved in random order. The INVALID KEY condition occurs when RELATIVE KEY contains an address pointing to a record that was deleted previously (see DELETE verb, following), or to an address beyond the boundaries of the file.

The WRITE, REWRITE, DELETE, and START statements parallel the ones discussed in Chapter 13 with respect to indexed files. Figures 14-5 and 14-6

```
Format 1 (for Sequential Access):
    READ file-name-1   [NEXT] RECORD   [INTO identifier-1]
        [AT END imperative-statement-1]
        [NOT AT END imperative-statement-2]
        [END-READ]
Format 2 (for Random Access):
    READ file-name-1   RECORD   [INTO identifier-1]
        [INVALID KEY imperative-statement-1]
        [NOT INVALID KEY imperative-statement-2]
        [END-READ]
```

FIGURE 14-4
GENERAL FORMATS FOR THE READ STATEMENT.

```
WRITE   record-name-1   [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-WRITE]

REWRITE   record-name-1   [FROM identifier-1]
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-REWRITE]

DELETE file-name-1 RECORD
    [INVALID KEY imperative-statement-1]
    [NOT INVALID KEY imperative-statement-2]
    [END-DELETE]
```

FIGURE 14-5
GENERAL FORMAT FOR THE WRITE, REWRITE, AND DELETE STATEMENTS.

```
                            ⎡      ⎧ IS EQUAL TO                       ⎫         ⎤
                            ⎢      ⎪ IS =                              ⎪         ⎥
                            ⎢      ⎪ IS GREATER THAN                   ⎪         ⎥
                            ⎢      ⎪ IS >                              ⎪         ⎥
START file-name-1           ⎢ KEY  ⎨ IS NOT LESS THAN                  ⎬ data-name-1 ⎥
                            ⎢      ⎪ IS NOT <                          ⎪         ⎥
                            ⎢      ⎪ IS GREATER THAN OR EQUAL TO       ⎪         ⎥
                            ⎣      ⎩ IS > =                            ⎭         ⎦
[INVALID KEY imperative-statement-1]
[NOT INVALID KEY imperative-statement-2]
[END-START]
```

FIGURE 14-6
GENERAL FORMAT FOR THE START VERB.

present the general format for these statements. In all cases, if the effective access mode is random, the RELATIVE KEY field must contain the record number of the record involved in the I/O operation. For instance, to DELETE a record, we must first determine its location address, put that address value in the RELATIVE KEY field, issue a READ command and then a DELETE command.

Sequential retrieval of records may be accomplished using the SEQUENTIAL or DYNAMIC access mode, the issuance of an OPEN INPUT instruction, and repetitive execution of READ or READ NEXT. Whenever sequential retrieval is desired from a point other than the beginning of a file, then the START command can be used. The same format applies as for indexed files, except that the starting record location is placed in the RELATIVE KEY field.

Review

1 The location address determined by a key-to-address routine is stored in the ________________ field.

RELATIVE KEY

2 For an indexed file, the RECORD KEY [is / is not] part of the original file record. For a relative file, the RELATIVE KEY [is / is not] part of the original file record.

is; is not

3 In the SELECT statement of the ENVIRONMENT DIVISION, the RELATIVE KEY clause is not needed and may be omitted if the file is to be created [sequentially / randomly].

sequentially

4 Records stored sequentially in a relative file [can / cannot] be accessed randomly.

can

5 Records stored randomly in a relative file [can / cannot] be accessed sequentially.

can

AN EXAMPLE OF CREATING A RELATIVE FILE

A source file is to be transferred to disk, organized as a relative file. For simplicity, we assume that there will be no more than 50 records. Each record contains an ITEM-NUMBER in columns 1–5 and an ITEM-NAME in columns 6–25. We

shall use ITEM-NUMBER as the identifier for each record and we shall compute disk addresses using this identifier and the prime number 47. By using the division remainder technique, we convert each ITEM-NUMBER value to a (relative) disk address in the range 1–50.

Figure 14-7 presents a program written to create such a relative file.

In the 010-MAIN-ROUTINE we perform 020-ZERO-DISK as many times as

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. RELATIVE-CREATE.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT DISK-FILE  ASSIGN TO DISK
                       ORGANIZATION IS RELATIVE
                       ACCESS MODE IS RANDOM
                       RELATIVE KEY IS LOCATION-ADDRESS.
*
     SELECT SOURCE-FILE  ASSIGN TO READER.
     SELECT PRINT-FILE   ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
*
 FD  DISK-FILE       LABEL RECORDS OMITTED
                     DATA RECORD IS DISK-RECORD
                     RECORD CONTAINS 26 CHARACTERS.
 01  DISK-RECORD.
     02 REC-STATUS-CODE      PIC 9.
         88 VACANT-RECORD                 VALUE ZERO.
         88 USED-RECORD                   VALUE 1.
     02 ITEM-NUMBER          PIC 9(5).
     02 ITEM-NAME            PIC X(20).
*
 FD  SOURCE-FILE     LABEL RECORDS OMITTED
                     DATA RECORD IS SOURCE-RECORD.
 01  SOURCE-RECORD.
     02 ITEM-NUMBER          PIC 9(5).
     02 ITEM-NAME            PIC X(20).
     02 FILLER               PIC X(55).
*
 FD  PRINT-FILE      LABEL RECORDS OMITTED
                     DATA RECORD IS PRINT-RECORD.
 01  PRINT-RECORD            PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-DATA-FLAG        PIC XXX      VALUE 'NO'.
     88 END-OF-SOURCE                     VALUE 'YES'.
*
 01  LOOP-FLAG               PIC XXX.
     88 FILE-IS-FULL                      VALUE 'YES'.
*
 01  LOCATION-ADDRESS        PIC 999.
 01  STARTING-ADDRESS        PIC 999.
 01  PRIME-NUMBER            PIC 99       VALUE 47.
 01  WORKFIELD               PIC S99999 USAGE COMP.
 01  QUOTIENT                PIC S999 USAGE COMP.
 01  MAX-NO-OF-LOCATIONS     PIC 999 VALUE 50.
 01  I                       PIC 999.
*
 01  ERROR-RECORD.
     02 FILLER               PIC X VALUE SPACE.
     02 ERROR-LOCATION       PIC ZZ999.
     02 FILLER               PIC XX VALUE SPACE.
     02 ERR-MESSAGE          PIC X(50).
```

FIGURE 14-7
SAMPLE PROGRAM FOR CREATING A RELATIVE FILE.

```
/
 PROCEDURE DIVISION.
*
 010-MAIN-ROUTINE.
     OPEN INPUT  SOURCE-FILE
     OPEN OUTPUT DISK-FILE
                 PRINT-FILE.
*
     PERFORM 020-ZERO-DISK
         VARYING I FROM 1 BY 1 UNTIL I > MAX-NO-OF-LOCATIONS.
*
     CLOSE DISK-FILE.
*
     OPEN I-O DISK-FILE
*
     PERFORM 030-READ-PROCESS-SOURCE
         UNTIL END-OF-SOURCE
               OR FILE-IS-FULL.
*
     CLOSE SOURCE-FILE
           PRINT-FILE
           DISK-FILE.
     STOP RUN.
*
 020-ZERO-DISK.
     MOVE I TO LOCATION-ADDRESS
     MOVE ZERO TO REC-STATUS-CODE
*
     WRITE DISK-RECORD
         INVALID KEY PERFORM 080-CANT-ACCESS.
*
 030-READ-PROCESS-SOURCE.
     READ SOURCE-FILE RECORD
            AT END MOVE 'YES' TO END-OF-DATA-FLAG.
     IF NOT END-OF-SOURCE
         PERFORM 040-LOAD-RECORD.
 040-LOAD-RECORD.
     PERFORM 050-RANDOMIZE-READ
     IF VACANT-RECORD
         NEXT SENTENCE
     ELSE
         MOVE LOCATION-ADDRESS TO STARTING-ADDRESS
         MOVE 'NO' TO LOOP-FLAG
         PERFORM 060-HANDLE-SYNONYMS
             UNTIL VACANT-RECORD
                   OR FILE-IS-FULL.
*
     IF FILE-IS-FULL
         NEXT SENTENCE
     ELSE
         MOVE 1 TO REC-STATUS-CODE
         MOVE ITEM-NUMBER OF SOURCE-RECORD
              TO ITEM-NUMBER OF DISK-RECORD
         MOVE ITEM-NAME OF SOURCE-RECORD
              TO ITEM-NAME OF DISK-RECORD
*
         REWRITE DISK-RECORD
              INVALID KEY PERFORM 080-CANT-ACCESS.
*
 050-RANDOMIZE-READ.
     MOVE ITEM-NUMBER OF SOURCE-RECORD TO WORKFIELD
     DIVIDE PRIME-NUMBER INTO WORKFIELD GIVING QUOTIENT
     COMPUTE LOCATION-ADDRESS =
         WORKFIELD - (PRIME-NUMBER * QUOTIENT) + 1
*
*COULD ALSO WRITE THE FOLLOWING INSTEAD OF COMPUTE:
*  DIVIDE PRIME-NUMBER INTO WORKFIELD GIVING QUOTIENT
*        REMAINDER LOCATION-ADDRESS
*  ADD 1 TO LOCATION-ADDRESS
```

FIGURE 14-7 *(Continued)*

SAMPLE PROGRAM FOR CREATING A RELATIVE FILE.

```
      *
           PERFORM 070-READ-DISK-REC.
      *
       060-HANDLE-SYNONYMS.
           ADD 1 TO LOCATION-ADDRESS
           IF LOCATION-ADDRESS > MAX-NO-OF-LOCATIONS
               SUBTRACT MAX-NO-OF-LOCATIONS FROM LOCATION-ADDRESS.
           IF LOCATION-ADDRESS = STARTING-ADDRESS
              MOVE ITEM-NUMBER OF DISK-RECORD TO ERROR-LOCATION
              MOVE 'FILE IS FULL; NEW RECORD IS' TO ERR-MESSAGE
              WRITE PRINT-RECORD FROM ERROR-RECORD AFTER 2 LINES
           ELSE
              PERFORM 070-READ-DISK-REC.
      *
       070-READ-DISK-REC.
             READ DISK-FILE RECORD
                INVALID KEY PERFORM 080-CANT-ACCESS.
       080-CANT-ACCESS.
           MOVE LOCATION-ADDRESS TO ERROR-LOCATION
           MOVE 'THIS RECORD LOCATION CAUSED INVALID KEY'
               TO ERR-MESSAGE.
           WRITE PRINT-RECORD FROM ERROR-RECORD AFTER 2.
```

FIGURE 14-7 *(Continued)*
SAMPLE PROGRAM FOR CREATING A RELATIVE FILE.

the value of MAX-NO-OF-LOCATIONS, which was set to 50 in the DATA DIVISION. The function of 020-ZERO-DISK is to initialize to zero the REC-STATUS-CODE of each record space, thereby indicating a vacant record space.

The main task in the 020-MAIN-ROUTINE is the repeated execution of 030-READ-PROCESS-SOURCE, whose function can be outlined as follows: We read each source record and compute its home-address (LOCATION-ADDRESS) in the 050-RANDOMIZE-READ paragraph. We then access (READ) the contents of the record specified by LOCATION-ADDRESS. If the space is vacant (IF VACANT-RECORD), we indicate that it no longer is vacant and we write the record there:

```
MOVE 1  TO REC-STATUS-CODE
MOVE ITEM-NUMBER OF SOURCE-RECORD
     TO ITEM-NUMBER OF DISK-RECORD
     (etc.)
```

If the home address computed in 050-RANDOMIZE-READ is not vacant, we proceed to search for a vacant space in the following consecutive locations. In order to prevent infinite looping, we use LOOP-FLAG and STARTING-ADDRESS. Let us consider an example. Suppose that a record transforms to location 40. We set LOOP-FLAG to "NO" and STARTING-ADDRESS to 40. If location 40 is already occupied, we increment LOCATION-ADDRESS to 41 and check to determine if that location is vacant. If it is not vacant, we continue searching for a vacant location by increments of 1. Notice that we avoid going beyond the file size (50) in 060-HANDLE-SYNONYMS by subtracting MAX-NO-OF-LOCATIONS FROM LOCATION-ADDRESS. This subtraction brings us to the beginning of the file, and we continue searching for a vacancy. In the event that the entire file is full, LOCATION-ADDRESS eventually would become equal to the STARTING-ADDRESS, indicating that we have come full circle. At that point, we set LOOP-FLAG to "YES" and we print an error message. The program would terminate because a "YES" in LOOP-FLAG is the condition-name FILE-IS-FULL, which causes termination in the 010-MAIN-ROUTINE as shown:

```
PERFORM 030-READ-PROCESS-SOURCE
    UNTIL END-OF-SOURCE
        OR FILE-IS-FULL.
```

AN EXAMPLE OF UPDATING A RELATIVE FILE

We illustrate the use of relative file organization by a sample program that updates the file created by the program in Figure 14-7. Figures 14-8 and 14-9 present the structure chart and the corresponding program for the update task.

Transaction records are submitted through a TRANS-FILE assigned to the card reader. In each transaction record, there is a code indicating the type of transaction:

```
01  TRANS-REC.
    02  TRANS-CODE              PIC 9.
        88  ADD-TRANS           VALUE ZERO.
        88  DELETE-TRANS        VALUE 1.
        88  MODIFY-TRANS        VALUE 2.
        88  ERROR-TRANS         VALUES 3 THRU 9.
    02  ITEM-NUMBER             PIC 9(5).
    02  ITEM-NAME               PIC X(20).
    02  FILLER                  PIC X(54).
```

An ADD-TRANS indicates the addition of a new record to the disk file, a DELETE-TRANS indicates the deletion of a record existing in the file, while a MODIFY-TRANS represents a change in the ITEM-NAME of the disk record.

It should be noted that disk records are specified by the following description:

```
01  DISK-RECORD.
    02  REC-STATUS-CODE         PIC 9.
        88  VACANT-RECORD       VALUE ZERO.
        88  USED-RECORD         VALUE 1.
        88  DELETED-RECORD      VALUE 2.
    02  ITEM-NUMBER             PIC 9(5).
    02  ITEM-NAME               PIC X(20).
```

When a record is deleted, the REC-STATUS-CODE for that record space is set equal to 2. On any subsequent occasion, we can identify the fact that the record space is available for a new record, and we ignore it when looking for synonyms, as explained in the first section of this chapter. For instance, suppose that a transaction record specifies deletion of an item-number. In the 600-DELETE-RECORD paragraph of Figure 14-9, we see that we PERFORM 300-RANDOMIZE-READ to compute the home address of the item record and then read the record at the home address. Then we say:

```
PERFORM 650-FIND-RECORD
        UNTIL RECORD-FOUND OR RECORD-NOT-FOUND
        OR INVALID-READ.
```

In the 650-FIND-RECORD paragraph, we first check to determine if the record space at the home address is occupied (IF USED-RECORD). If it is occupied, we check to determine if the ITEM-NUMBER in the transaction record matches the one in the disk record, in which case that is the record to be deleted. If the record space is occupied but does not contain the record that we want, then the 650-FIND-RECORD paragraph PERFORMs 450-HANDLE-SYNONYMS. As a result, the next disk record is read and we repeat the process under control of the PERFORM 650-FIND-RECORD UNTIL . . . loop in the 600-DELETE-RECORD paragraph.

Continuing our analysis of the 650-FIND-RECORD paragraph, note that we say IF DELETED-RECORD PERFORM 450-HANDLE-SYNONYMS PER-

FORM 700-CHECK-IF-FULL-CIRCLE. If there is a deleted record, we simply PERFORM 450-HANDLE-SYNONYMS in order to move to the next disk record space on the relative file. The 700-CHECK-IF-FULL-CIRCLE procedure prevents us from searching the entire file repeatedly for a nonexisting record, in the case where all records are occupied or vacant from deletions only. Notice that the 650-FIND-RECORD terminates (MOVE 1 TO RECORD-FOUND-FLAG) when a vacant record is encountered (IF VACANT-RECORD), since such an occurrence indicates that the record does not exist.

The program is substantially self-documenting, and the student should be able to review it and follow the details. The overall task is typical of most such update programs, and the sample program can be used as the basic structure for most of them.

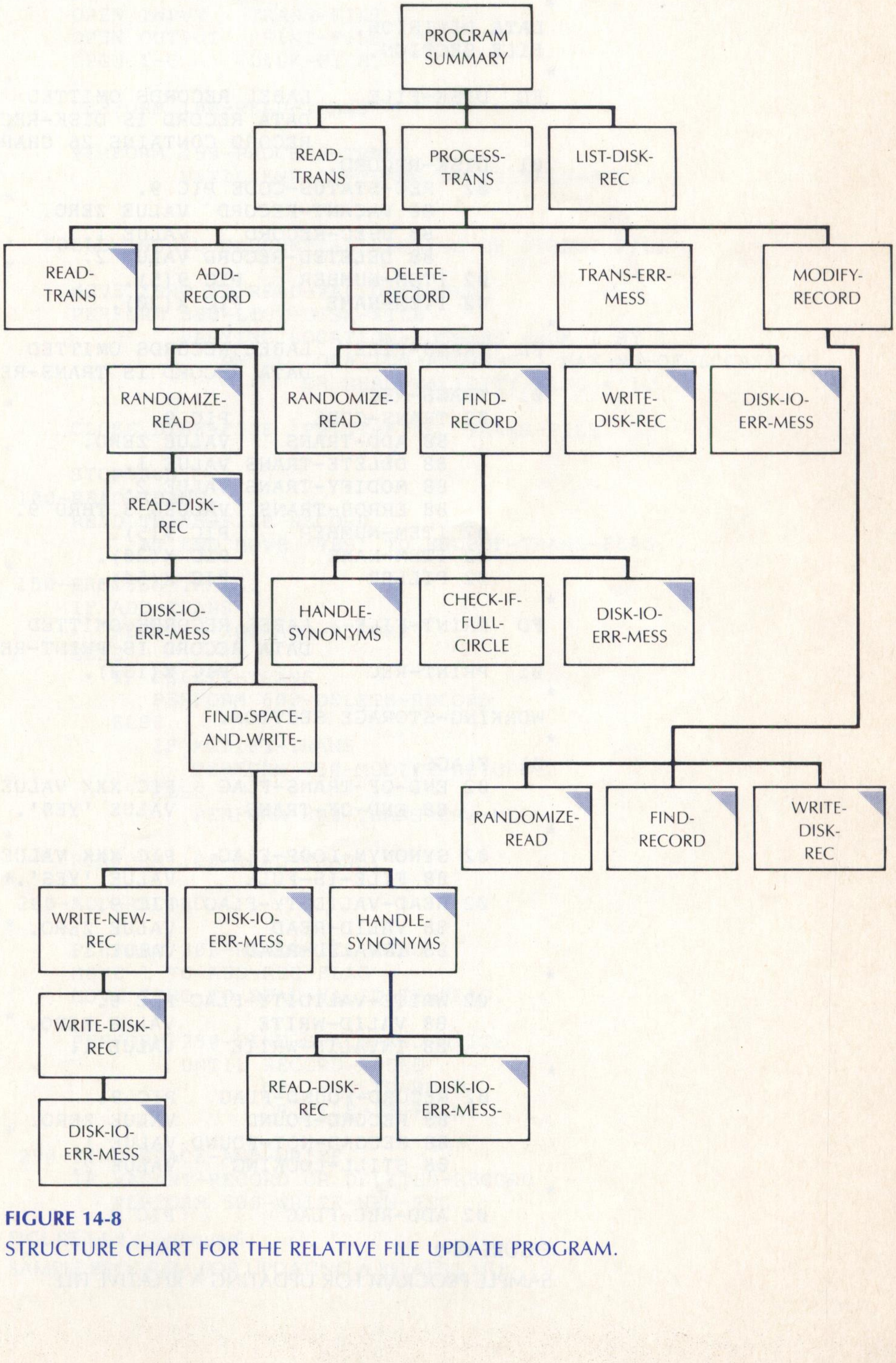


FIGURE 14-8
STRUCTURE CHART FOR THE RELATIVE FILE UPDATE PROGRAM.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID. REL-UPDATE.
*
 ENVIRONMENT DIVISION.
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT DISK-FILE  ASSIGN TO DISK
                       ORGANIZATION IS RELATIVE
                       ACCESS MODE IS RANDOM
                       RELATIVE KEY IS LOCATION-ADDRESS.
*
     SELECT TRANS-FILE ASSIGN TO READER.
     SELECT PRINT-FILE ASSIGN TO PRINTER.
*
 DATA DIVISION.
 FILE SECTION.
*
 FD  DISK-FILE      LABEL RECORDS OMITTED
                    DATA RECORD IS DISK-RECORD
                    RECORD CONTAINS 26 CHARACTERS.
 01  DISK-RECORD.
     02  REC-STATUS-CODE PIC 9.
         88 VACANT-RECORD  VALUE ZERO.
         88 USED-RECORD    VALUE 1.
         88 DELETED-RECORD VALUE 2.
     02 ITEM-NUMBER      PIC 9(5).
     02 ITEM-NAME        PIC X(20).
*
 FD  TRANS-FILE     LABEL RECORDS OMITTED
                    DATA RECORD IS TRANS-REC.
 01  TRANS-REC.
     02 TRANS-CODE          PIC 9.
        88 ADD-TRANS        VALUE ZERO.
        88 DELETE-TRANS     VALUE 1.
        88 MODIFY-TRANS     VALUE 2.
        88 ERROR-TRANS      VALUES 3 THRU 9.
     02 ITEM-NUMBER         PIC 9(5).
     02 ITEM-NAME           PIC X(20).
     02 FILLER              PIC X(54).
*
 FD  PRINT-FILE     LABEL RECORDS OMITTED
                    DATA RECORD IS PRINT-REC.
 01  PRINT-REC              PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  FLAGS.
     02 END-OF-TRANS-FLAG     PIC XXX VALUE 'NO'.
        88 END-OF-TRANS       VALUE 'YES'.
*
     02 SYNONYM-LOOP-FLAG     PIC XXX VALUE 'NO'.
        88 FILE-IS-FULL       VALUE 'YES'.
     02 READ-VALIDITY-FLAG    PIC 9.
        88 VALID-READ         VALUE ZERO.
        88 INVALID-READ       VALUE 1.
*
     02 WRITE-VALIDITY-FLAG   PIC 9.
        88 VALID-WRITE        VALUE ZERO.
        88 INVALID-WRITE      VALUE 1.
*
     02 RECORD-FOUND-FLAG     PIC 9.
        88 RECORD-FOUND       VALUE ZERO.
        88 RECORD-NOT-FOUND   VALUE 1.
        88 STILL-LOOKING      VALUE 2.
*
     02 ADD-REC-FLAG          PIC 9.
```

FIGURE 14-9

SAMPLE PROGRAM FOR UPDATING A RELATIVE FILE.

```
          88 RECORD-ADDED       VALUE ZERO.
          88 RECORD-NOT-ADDED VALUE 1.
      *
       01  LOCATION-ADDRESS     PIC 9(3).
       01  MAX-NO-OF-LOCATIONS PIC 9(3) VALUE 50.
       01  STARTING-ADDRESS     PIC 9(3).
       01  PRIME-NUMBER         PIC 99 VALUE 57.
       01  WORKFIELD            PIC S9(5).
       01  QUOTIENT             PIC S9(3).
      *
       01  ERROR-RECORD.
           02 FILLER          PIC X VALUE SPACE.
           02 ERROR-LOCATION  PIC ZZ999.
           02 ERR-MESSAGE     PIC X(50).
      /
       PROCEDURE DIVISION.
      *
       050-PROGRAM-SUMMARY.
           OPEN INPUT   TRANS-FILE
           OPEN OUTPUT  PRINT-FILE
           OPEN I-O     DISK-FILE.
      *
           PERFORM 100-READ-TRANS
      *
           PERFORM 150-PROCESS-TRANS
               UNTIL END-OF-TRANS OR FILE-IS-FULL.
      *
      *
      *  FOLLOWING IS SEQUENTIAL LISTING OF DIRECT FILE
      *
           MOVE ZERO TO READ-VALIDITY-FLAG
           PERFORM 800-LIST-DISK-FILE
               VARYING LOCATION-ADDRESS FROM 1 BY 1
                  UNTIL LOCATION-ADDRESS > MAX-NO-OF-LOCATIONS
                        OR READ-VALIDITY-FLAG = 1.

           CLOSE DISK-FILE  PRINT-FILE  TRANS-FILE
      *
           STOP RUN.
       100-READ-TRANS.
           READ TRANS-FILE RECORD
               AT END MOVE 'YES' TO END-OF-TRANS-FLAG.
      *
       150-PROCESS-TRANS.
           IF ADD-TRANS
              PERFORM 200-ADD-RECORD
           ELSE
              IF DELETE-TRANS
                 PERFORM 600-DELETE-RECORD
              ELSE
                 IF MODIFY-TRANS
                    PERFORM 750-MODIFY-RECORD
                 ELSE
                    PERFORM 800-TRANS-ERR-MESS.
      *
           PERFORM 100-READ-TRANS.
      *
       200-ADD-RECORD.
      *
           PERFORM 300-RANDOMIZE-READ
           MOVE 1 TO ADD-REC-FLAG
           MOVE ZERO TO READ-VALIDITY-FLAG.
      *
           PERFORM 250-FIND-SPACE-AND-WRITE
               UNTIL RECORD-ADDED
                     OR INVALID-READ
                     OR FILE-IS-FULL.
      *
       250-FIND-SPACE-AND-WRITE.
           IF VACANT-RECORD OR DELETED-RECORD
              PERFORM 500-WRITE-NEW-REC
```

FIGURE 14-9 *(Continued)*
SAMPLE PROGRAM FOR UPDATING A RELATIVE FILE.

```
            MOVE ZERO TO ADD-REC-FLAG
         ELSE
            IF ITEM-NUMBER OF TRANS-REC = ITEM-NUMBER OF DISK-RECORD
               MOVE 'ATTEMPT TO ADD DUPLICATE RECORD' TO ERR-MESSAGE
               PERFORM 400-DISK-IO-ERR-MESS
               MOVE 1 TO READ-VALIDITY-FLAG
            ELSE
               PERFORM 450-HANDLE-SYNONYMS.
*
 300-RANDOMIZE-READ.
     MOVE ITEM-NUMBER OF TRANS-REC TO WORKFIELD
     DIVIDE PRIME-NUMBER INTO WORKFIELD GIVING QUOTIENT
     COMPUTE LOCATION-ADDRESS =
             WORKFIELD - (QUOTIENT * PRIME-NUMBER) + 1
*
     MOVE 'NO' TO SYNONYM-LOOP-FLAG
     MOVE LOCATION-ADDRESS TO STARTING-ADDRESS
*
     PERFORM 350-READ-DISK-REC.
 350-READ-DISK-REC.
     MOVE ZERO TO READ-VALIDITY-FLAG
     READ DISK-FILE RECORD
        INVALID KEY MOVE 1 TO READ-VALIDITY-FLAG.
*
     IF INVALID-READ
        MOVE 'THIS RECORD LOCATION CANNOT BE READ'
            TO ERR-MESSAGE
        PERFORM 400-DISK-IO-ERR-MESS.
*
 400-DISK-IO-ERR-MESS.
     MOVE LOCATION-ADDRESS TO ERROR-LOCATION
     WRITE PRINT-REC FROM ERROR-RECORD AFTER 2 LINES
     MOVE SPACES TO ERROR-RECORD
     MOVE 'TRANSACTION RECORD IS' TO ERR-MESSAGE
     WRITE PRINT-REC FROM ERROR-RECORD AFTER 2 LINES
     MOVE TRANS-REC TO ERR-MESSAGE
     WRITE PRINT-REC FROM ERROR-RECORD AFTER 2 LINES.
*
 450-HANDLE-SYNONYMS.
     MOVE 'NO' TO SYNONYM-LOOP-FLAG
     ADD 1 TO LOCATION-ADDRESS
     IF LOCATION-ADDRESS GREATER THAN MAX-NO-OF-LOCATIONS
        MOVE 1 TO LOCATION-ADDRESS.
*
     IF LOCATION-ADDRESS = STARTING-ADDRESS
        MOVE 'ENTIRE FILE READ FULL CIRCLE' TO ERR-MESSAGE
        PERFORM 400-DISK-IO-ERR-MESS
*
        MOVE 'YES' TO SYNONYM-LOOP-FLAG
     ELSE
        PERFORM 350-READ-DISK-REC.
*
 500-WRITE-NEW-REC.
     MOVE 1 TO REC-STATUS-CODE
     MOVE ITEM-NUMBER OF TRANS-REC
          TO ITEM-NUMBER OF DISK-RECORD
     MOVE ITEM-NAME OF TRANS-REC
          TO ITEM-NAME OF DISK-RECORD

     PERFORM 550-WRITE-DISK-REC.
*
 550-WRITE-DISK-REC.
     MOVE ZERO TO WRITE-VALIDITY-FLAG
     REWRITE DISK-RECORD
             INVALID KEY MOVE 1 TO WRITE-VALIDITY-FLAG.
     IF INVALID-WRITE
        MOVE 'THIS RECORD LOCATION CANNOT BE WRITTEN'
            TO ERR-MESSAGE
        PERFORM 400-DISK-IO-ERR-MESS.
 600-DELETE-RECORD.
     MOVE 2 TO RECORD-FOUND-FLAG
     MOVE ZERO TO READ-VALIDITY-FLAG
```

FIGURE 14-9 *(Continued)*
SAMPLE PROGRAM FOR UPDATING A RELATIVE FILE.

```
       *
           PERFORM 300-RANDOMIZE-READ
       *
           PERFORM 650-FIND-RECORD
               UNTIL RECORD-FOUND
                     OR RECORD-NOT-FOUND
                     OR INVALID-READ.
       *
           IF RECORD-FOUND
              MOVE 1 TO REC-STATUS-CODE
              PERFORM 550-WRITE-DISK-REC
           ELSE
              IF RECORD-NOT-FOUND
                  MOVE 'ATTEMPT TO DELETE NONEXISTENT RECORD'
                       TO ERR-MESSAGE
                  PERFORM 400-DISK-IO-ERR-MESS.
       *
        650-FIND-RECORD.
           IF USED-RECORD
              IF ITEM-NUMBER OF TRANS-REC = ITEM-NUMBER OF DISK-RECORD
                 MOVE ZERO TO RECORD-FOUND-FLAG
              ELSE
                  PERFORM 450-HANDLE-SYNONYMS
                  PERFORM 700-CHECK-IF-FULL-CIRCLE
           ELSE
              IF DELETED-RECORD
                 PERFORM 450-HANDLE-SYNONYMS
                 PERFORM 700-CHECK-IF-FULL-CIRCLE
              ELSE
                 IF VACANT-RECORD
                    MOVE 1 TO RECORD-FOUND-FLAG
                 ELSE
                    MOVE 1 TO RECORD-FOUND-FLAG
                    MOVE 'STATUS CODE OF DISK-REC IS INVALID'
                      TO ERR-MESSAGE
                    PERFORM 400-DISK-IO-ERR-MESS.
       *
        700-CHECK-IF-FULL-CIRCLE.
           IF FILE-IS-FULL
              MOVE 'CAME FULL CIRCLE' TO ERR-MESSAGE
              PERFORM 400-DISK-IO-ERR-MESS
              MOVE 1 TO RECORD-FOUND-FLAG.
       *
        750-MODIFY-RECORD.
           MOVE 2 TO RECORD-FOUND-FLAG
           MOVE ZERO TO READ-VALIDITY-FLAG
           PERFORM 300-RANDOMIZE-READ.
           PERFORM 650-FIND-RECORD
               UNTIL RECORD-FOUND
                     OR RECORD-NOT-FOUND
                     OR INVALID-READ.
       *
           IF RECORD-FOUND
              MOVE ITEM-NAME OF TRANS-REC TO ITEM-NAME OF DISK-RECORD
              PERFORM 550-WRITE-DISK-REC
           ELSE
              IF RECORD-NOT-FOUND
                  MOVE 'ATTEMPT TO MODIFY NONEXISTENT RECORD'
                       TO ERR-MESSAGE
                  PERFORM 400-DISK-IO-ERR-MESS.
       *
        800-TRANS-ERR-MESS.
           MOVE SPACES TO ERROR-RECORD
           MOVE 'THIS TRANSACTION HAS INVALID-CODE' TO ERR-MESSAGE
           WRITE PRINT-REC FROM ERROR-RECORD AFTER 2 LINES
           MOVE TRANS-REC TO ERR-MESSAGE
           WRITE PRINT-REC FROM ERROR-RECORD AFTER 2 LINES.
       *
        850-LIST-DISK-FILE.
           READ DISK-FILE RECORD
             INVALID KEY MOVE 1 TO READ-VALIDITY-FLAG.
           IF READ-VALIDITY-FLAG = ZERO
              WRITE PRINT-REC FROM DISK-RECORD AFTER 1.
```

FIGURE 14-9 *(Continued)*

SAMPLE PROGRAM FOR UPDATING A RELATIVE FILE.

EXERCISES

14.1 Do a digit analysis of the following set of customer account numbers: 8023, 9178, 9034, 8187, 8056, 9162, 9019.

14.2 Based on the digit analysis in Exercise 14.1, describe a key-to-address transformation method for a file that consists of 100 customer accounts. Demonstrate the procedure by computing the address for account numbers 8023 and 3456.

14.3 Outline what changes would be needed in the sample program for creating a relative file in Figure 14-7 to guarantee that there will be no duplicate records in the file. A duplicate record is one that has the same key as another record in the file.

14.4 Outline what changes would be needed in the sample program for creating and updating a relative file in Figure 14-9 to guarantee that there will be no duplicate records in the file. A duplicate record is one that has the same key as another record in the file.

14.5 Refer to Exercise 9.4 for the program description.

a Create the master file as a relative file, using the part number as the RELATIVE KEY.

b Update the master file on a random basis, using the part number as the RELATIVE KEY.

14.6 A manufacturer of three product classes has a sales force consisting of 100 salespeople, each person assigned a unique salesperson number of five digits. Salespeople are paid on commission, receiving monthly commission benefits as well as an annual bonus based on monthly performance. We want to maintain commission data for each salesperson, by product class and by month, on a disk file.

a Create a relative file that will contain a record of the commission data for each salesperson in the following form. Salesperson number will serve for the RELATIVE KEY.

FIELD	FIELD SIZE
Salesperson number	5 digits
Salesperson-name	
Last name	15 characters
First name	10 characters
Middle initial	1 character
Commission totals by product class (3 classes) and by month (12 months).	Each total can be as large as 999,999.99. Note: There will be 36 totals.

The file is created by reading one record per salesperson, containing the salesperson-number and salesperson-name fields. After the randomizing technique has been employed to determine the disk location, all commission totals (36 fields) are set to zero. Then the record is written on the disk.

b Update this file, using sales transaction data. Use the salesperson number as the RELATIVE KEY.

The transaction records have the following layout:

CARD COLUMNS	FIELD
1–5	Salesperson number
6	Commission code (based on product class)
	1 = 0.02 of sales
	2 = 0.03 of sales
	3 = 0.05 of sales
7–8	Month code (from 01 to 12)
9–14	Sales value in dollars and cents

Assume that the transaction records are sorted by salesperson number; therefore, we need to access the relevant master record only once for each set of records corresponding to one salesperson. Of course, we may have transactions for only *some* of the salespeople.

As each salesperson is processed, we want to print on the printer a report, as follows:

Current Commission Data			
		TOTALS	
SALESPERSON NUMBER	SALESPERSON-NAME	THIS MONTH	YEAR-TO-DATE
12345	LAST, FIRST, M.	$ 870.35	$18,562.40
24966	LAST, FIRST, M.	1020.20	12,112.96
.	.	.	.
.	.	.	.
.	.	.	.

In other words, we want to accumulate the commissions, regardless of product class, for the current month; as well as the year-to-date totals for all months through the present one.

15 SUBPROGRAMS AND NESTED PROGRAMS

- INTRODUCTION
- CALLING AND CALLED PROGRAMS
- SAMPLE PROGRAM WITH A SUBPROGRAM
- COBOL STATEMENTS FOR SUBPROGRAMS
- NESTED COBOL PROGRAMS
- EXERCISES

INTRODUCTION

Thus far we have utilized the PERFORM verb as the basic control mechanism for implementing modular program structure. Still, it is often desirable to program a task in terms of one main program and one or more subprograms. In such a structure, the main program is the executable program. Subprograms can be written and compiled independently, but they can be executed only in conjunction with a main program. There are three basic reasons why subprograms are desirable:

1. Whenever a task is either too large for one person or the time available requires the formation of a project team, subprograms are a natural way of partitioning one task among several persons. Because subprograms can be compiled independently, each team member can work individually to develop and test a portion of the total task. Communication among the team members is limited to brief coordinative activities assigned to a chief programmer who is responsible for the overall project design and for effective and efficient interfacing between subtasks partitioned out as subprograms.
2. There is a frequent need to incorporate the same task into more than one program. In such a case, a subprogram that is written and tested once can be recorded in a program library and can be used by several programs, thus avoiding the "reinvention-of-the-wheel."
3. When using subprograms we can identify the data items specifically involved with the functions performed by each individual subprogram. In contrast, there is only one DATA DIVISION in each program, and it contains a description of the data for all modules in the program, as a group.

When using subprograms, there is one main program and one or more subprograms. The main program initiates and controls execution of the entire job, including the execution of subprograms, and eventually terminates the job. A given subprogram may be called into execution by the main program or it may be called by another subprogram; however, in a given program there must be at least one call issued by the main program, and that must be the first call. After that point, subprograms may call each other—although they cannot call themselves (recursion is not allowed). You may have noticed the use of the word "call." It is standard terminology in reference to subprogram execution, and it is implemented in COBOL through the verb CALL.

Review

1 In lieu of using the PERFORM verb, modular program structure can be implemented by writing separate ________________.

subprograms

2 The use of subprograms makes the partitioning of a programming task among several individuals [easier/more difficult].

easier

3 A subprogram [can/cannot] be used easily in conjunction with different programs.

can

4 When a subprogram is written, the data involved in that subprogram are described specifically in the DATA DIVISION of the [main program/subprogram].

subprogram

CALLING AND CALLED PROGRAMS

Whenever subprograms are used, there is one so-called *main program* and one or more *subprograms*. The main program is the executable module; it starts the job and it terminates the job (with a STOP RUN statement). Somewhere in the main program there will be a CALL statement which will refer to a subprogram by its name. The CALL statement operates just like a PERFORM statement. If we write CALL 'SUB', we are in essence saying PERFORM SUB. However, there are differences, as well as similarities, between CALL and PERFORM.

The PERFORM statement references a paragraph or a section within the program. The CALL statement references a subprogram, and subprograms are separate, whole programs. So in essence, a CALL allows us to say 'execute another program.' Just as is the case with PERFORM, when the object of the PERFORM has been executed, control returns to the statement following the PERFORM. However, in subprograms we use a special verb, EXIT PROGRAM, which controls the return to the statement after the CALL. In the case of PERFORM, you will recall, we return when the physical end of the performed paragraph or section is reached.

Also, just as is the case with PERFORM, a called subprogram may CALL

another subprogram. For example, we can say PERFORM ABC and paragraph ABC may contain a PERFORM XYZ statement. Similarly, we can have:

```
Main Program:
      .
      .
    CALL 'SUB-A'
      .
      .
    CALL 'SUB-B'
      .
      .
    STOP RUN.

Subprogram SUB-A:
      .
      .
    CALL 'SUB-B'
      .
      .
    EXIT PROGRAM.
      .
      .
      .

Subprogram SUB-B
      .
      .
    EXIT PROGRAM.
      .
```

In the above illustration, the main program executes a subprogram called SUB-A which, in turn, executes another subprogram called SUB-B. In the above example we can see that a subprogram can be either a *called* or a *calling* program. Of course, the main program can only be a calling program; it is not permitted to have a subprogram call the main program.

Notice that subprogram SUB-B is called both by the main program and the SUB-A subprogram in the example above. Just as is the case with the PERFORM, a given subprogram may be called from more than one program.

As a final point, a principal difference between CALL and PERFORM is that a calling and a called program must, somehow, share common data. Since the PERFORM refers to the same program, all data have been defined in the DATA DIVISION. In the case of CALL, there are two programs involved and we must somehow *link* data that apply to both programs. Such a linkage is accomplished by specifying a list of *arguments* in the CALL statement and a parallel list in the PROCEDURE DIVISION header of the subprogram. Further, a special LINKAGE SECTION in the DATA DIVISION of a subprogram is used in order to define the data involved in both programs. Figure 15–1 shows an outline of the relevant portions of a main program and a subprogram for the following task.

CALLING PROGRAM	CALLED PROGRAM
IDENTIFICATION DIVISION. PROGRAM-ID. MAINPROG. . . . DATA DIVISION. . . . 01 A PIC 99. 01 B PIC 99. 01 A-PLUS-B PIC 999. 01 STATUS-CODE PIC X(4). . . . PROCEDURE DIVISION. CALL 'ADDER' USING A, B, A-PLUS-B, STATUS-CODE IF STATUS-CODE = 'O.K.' . . . ELSE . . .	IDENTIFICATION DIVISION. PROGRAM-ID. ADDER. . . . DATA DIVISION. . . . LINKAGE SECTION. 01 STATUS-CODE PIC X(4). 01 A PIC 99. 01 B PIC 99. 01 A-PLUS-B PIC 999. PROCEDURE DIVISION USING A, B, A-PLUS-B, STATUS-CODE. ADD-A-AND-B. IF A IS NUMERIC AND B IS NUMERIC MOVE 'O.K.' TO STATUS-CODE ADD A B GIVING A-PLUS-B ELSE MOVE SPACES TO STATUS-CODE. RETURN-TO-CALL. EXIT PROGRAM.

FIGURE 15-1
SAMPLE CALLING AND CALLED PROGRAM OUTLINE.

The main program calls a subprogram and gives it two values, contained in fields A and B. The subprogram receives these two values, checks to see if they are numeric, and if so, forms their sum in A-PLUS-B and puts 'O.K.' in STATUS-CODE. If the input data A and B are not numeric, the subprogram moves spaces to STATUS-CODE.

The example in Figure 15-1 illustrates a case in which the names used in the calling and the called program are the same. This need not be the case. The CALL statement passes to the called program the address of the data-names in the USING clause of the CALL. The subprogram then equates these addresses with the data names in the PROCEDURE DIVISION USING... clause, in corresponding order from first to last. The following revision of the relevant portion of the subprogram outline in Figure 15–1 illustrates the point:

```
.
.
.
LINKAGE SECTION.
01  X  PIC  99.
01  Y  PIC  99.
01  Z  PIC  999.
01  STATUS-CODE PIC X(4).
PROCEDURE DIVISION USING X, Y, Z, STATUS-CODE.
ADD-TWO-NUMBERS.
        IF X IS NUMERIC AND Y IS NUMERIC
```

```
        MOVE 'O.K.' TO STATUS-CODE
        ADD X Y GIVING Z
    ELSE
        MOVE SPACES TO STATUS-CODE.
```

By the revision above, what is called A in the calling program is called X in the subprogram, and so on. Thus, the author of the subprogram need not know what data-names are used in the calling program, and vice versa. What is important is the order of the arguments in the CALL . . . USING . . . specification in the calling program and the PROCEDURE DIVISION USING . . . specification in the called program. Further, notice that in the LINKAGE SECTION the order of data-names is irrelevant. However, their PIC description is important. In the above example, A-PLUS-B is the third item in the CALL argument list, and has a PIC 999 in the calling program. The third item in the PROCEDURE DIVISION USING . . . list is Z. It follows that Z must be an item in the LINKAGE SECTION and it must have a PIC 999 in order to process the data correctly.

Review

1 When subprograms are used, there must be one and only one __________ program but there may be several __________.

main; subprograms

2 When subprograms are used, at least one CALL statement is always included in the [main program / each subprogram].

main program

3 Program execution is returned from a subprogram to the statement after the CALL verb by use of the special verb in the subprogram, __________.

EXIT PROGRAM

4 A called subprogram [can / cannot] itself include a CALL statement that results in execution of another subprogram.

can

5 A called subprogram [can / cannot] itself include a CALL statement that results in execution of the main program.

cannot

6 A given subprogram [can / cannot] be called into execution from more than one other program.

can

7 Considering a difference between the CALL and PERFORM verbs, only one DATA DIVISION is involved in the context of using the __________ verb.

PERFORM

8 Because each subprogram has its own DATA DIVISION, and yet there are data that are to be used in both the main program and subprogram, the DATA DIVISION of the subprogram must include a special __________ SECTION to identify such data.

LINKAGE

9 For data that are used in both a main program and subprogram, the data-names [must / need not] be the same in the two programs.

need not

10 Since the data-names for the data items shared by the main program and subprogram need not be the same, what is important is the respective ________________ of the data-names in the CALL . . . USING statement of the calling program and the PROCEDURE DIVISION USING . . . statement of the called program.

order

11 For shared data items, the order of the data-names in the LINKAGE SECTION [is / is not] relevant.

is not

12 For shared data items, what is important in the LINKAGE SECTION is the ________________ description for each data-name.

PIC (*or* PICTURE)

SAMPLE PROGRAM WITH A SUBPROGRAM

Figures 15-2 and 15-3 illustrate a main program and a subprogram. These sample programs were written as follows:

MAIN PROGRAM. The main program reads source records containing sales data and stores them in a table. In the process, the program counts the number of data items, up to a maximum of 100 items. When all the data have been input, the main program calls a subprogram and provides the subprogram with the table of sales data and the number of items. If the indication from the subprogram is that there were no errors, the main program prints the data processed by the subprogram; otherwise, the main program sets the results data to zero to indicate the presence of errors.

SUBPROGRAM. This program finds the minimum and maximum value in the table of sales data and computes the average. If a SIZE ERROR condition occurs during accumulation of the total sales value, then the subprogram indicates the occurrence of such an error by putting spaces in a status field and then returns to the calling (main) program. If no error occurs, then the subprogram places the literal value 'O.K.' in the status field and returns to the main program.

```
 IDENTIFICATION DIVISION.
 PROGRAM-ID.  SALESTATS.
*
 ENVIRONMENT DIVISION.
*
 CONFIGURATION SECTION.
 SOURCE-COMPUTER. ABC-480.
 OBJECT-COMPUTER. ABC-480.
*
 INPUT-OUTPUT SECTION.
 FILE-CONTROL.
     SELECT SOURCE-FILE    ASSIGN TO S-SOURCE.
     SELECT OUTPUT-FILE    ASSIGN TO S-PRINTER.
*
 DATA DIVISION.
*
 FILE SECTION.
*
 FD  SOURCE-FILE            LABEL RECORDS STANDARD
                            DATA RECORD IS SOURCE-REC.
 01  SOURCE-REC.
     02  SALES-AMOUNT       PIC 9(4)V99.
     02  FILLER             PIC X(74).
```

FIGURE 15-2

SAMPLE MAIN PROGRAM.

```
*
 FD  OUTPUT-FILE                 LABEL RECORDS OMITTED
                                 DATA RECORD IS OUTPUT-REC.
 01  OUTPUT-REC                  PIC X(132).
*
 WORKING-STORAGE SECTION.
*
 01  END-OF-FILE-TEST            PIC XXX VALUE 'NO '.
     88 END-OF-FILE              VALUE 'YES'.
*
 01  COMPLETION-CODE             PIC X(4).
*
 01  NO-OF-RECS                  PIC 9(3) VALUE ZERO.
 01  RECORD-COUNTERS.
     02  N-GOOD                  PIC 9(3) VALUE ZERO.
     02  N-TOTAL                 PIC 9(3) VALUE ZERO.
*
 01   SALES-TABLE.
      02   SALES  OCCURS 100 TIMES PIC 9(4)V99.
*
 01   SALES-STATS.
     02  MIN                     PIC 9(4)V99.
     02  MAX                     PIC 9(4)V99.
     02  AVG                     PIC 9(4)V99.
 01  WS-OUTPUT-REC.
     02  FILLER                  PIC X(6) VALUE 'MIN = '.
     02  MIN-OUT                 PIC Z(4)9.99.
     02  FILLER                  PIC X(8) VALUE '   MAX = '.
     02  MAX-OUT                 PIC Z(4)9.99.
     02  FILLER                  PIC X(8) VALUE '   AVG = '.
     02  AVG-OUT                 PIC Z(4)9.99.
     02  FILLER                  PIC X(6) VALUE '   N = '.
     02  N-OUT                   PIC ZZ9.
 PROCEDURE DIVISION.
*
 PROGRAM-SUMMARY.
     OPEN INPUT SOURCE-FILE  OUTPUT OUTPUT-FILE
     PERFORM READ-SOURCE-RECORDS
*
     IF N-GOOD > ZERO
        MOVE N-GOOD TO NO-OF-RECS
        CALL 'STATS' USING NO-OF-RECS,
                           SALES-TABLE,
                           SALES-STATS,
                           COMPLETION-CODE
        IF COMPLETION-CODE = 'O.K.'
            NEXT SENTENCE
        ELSE
            MOVE ZEROS TO SALES-STATS
     ELSE
        MOVE ZEROS TO SALES-STATS.
*
     PERFORM PRINT-STATS
*
     CLOSE SOURCE-FILE OUTPUT-FILE
     STOP RUN.
*
 READ-SOURCE-RECORDS.
     PERFORM READ-SOURCE
     PERFORM STORE-SALES
         UNTIL  END-OF-FILE  OR N-GOOD = 100.
*
 READ-SOURCE.
     READ SOURCE-FILE AT END MOVE 'YES' TO END-OF-FILE-TEST.
*
 STORE-SALES.
     ADD 1 TO N-TOTAL
     IF SALES-AMOUNT IS NUMERIC
        ADD 1 TO N-GOOD
        MOVE SALES-AMOUNT TO SALES (N-GOOD)
```

FIGURE 15-2 *(Continued)*
SAMPLE MAIN PROGRAM.

```
    ELSE
        STRING 'RECORD ' N-TOTAL ' CONTAINS NONNUMERIC DATA: '
                      SALES-AMOUNT
                      DELIMITED BY SIZE
               INTO OUTPUT-REC
        WRITE OUTPUT-REC.
    PERFORM READ-SOURCE.
*
PRINT-STATS.
    MOVE MIN TO MIN-OUT
    MOVE MAX TO MAX-OUT
    MOVE AVG TO AVG-OUT
    MOVE NO-OF-RECS TO N-OUT
    WRITE OUTPUT-REC FROM WS-OUTPUT-REC.
```

FIGURE 15-2 (*Continued*)

SAMPLE MAIN PROGRAM.

```
IDENTIFICATION DIVISION.
PROGRAM-ID.  STATS.
*
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
*
DATA DIVISION.
WORKING-STORAGE SECTION.
01  SALES-TOTAL                  PIC 9(5)V99 VALUE ZERO.
01  I                            PIC 999.
*
LINKAGE SECTION.
*
01  NO-OF-VALUES                 PIC 9(3).
01  INPUT-TABLE.
    02  DATA-VALUE  OCCURS 100 TIMES PIC 9(4)V99.
*
01  OUTPUT-STATS.
    02  MIN                      PIC 9(4)V99.
    02  MAX                      PIC 9(4)V99.
    02  AVG                      PIC 9(4)V99.
01  STATUS-CODE                  PIC X(4).
*
PROCEDURE DIVISION USING NO-OF-VALUES, INPUT-TABLE
                         OUTPUT-STATS, STATUS-CODE.
*
PROGRAM-SUMMARY.
    MOVE 'O.K.' TO STATUS-CODE
    MOVE DATA-VALUE (1) TO MIN, MAX, SALES-TOTAL.
    PERFORM STATISTICS VARYING I FROM 1 BY 1
                         UNTIL I > NO-OF-VALUES
                         OR STATUS-CODE = SPACES.
    DIVIDE SALES-TOTAL BY NO-OF-VALUES GIVING AVG
          ON SIZE ERROR MOVE SPACES TO STATUS-CODE.
*
RETURN-TO-CALL.
    EXIT PROGRAM.
*
STATISTICS.
    IF MIN > DATA-VALUE (I)
       MOVE DATA-VALUE (I) TO MIN.
    IF MAX < DATA-VALUE (I)
       MOVE DATA-VALUE (I) TO MAX.
    ADD DATA-VALUE (I) TO SALES-TOTAL
        ON SIZE ERROR MOVE SPACES TO STATUS-CODE.
```

FIGURE 15-3

SAMPLE SUBPROGRAM.

Notice that the list of arguments includes a table, and that different names have been used for some of the arguments in the main program and the subprogram. The corresponding arguments are illustrated in the following arrangement:

```
CALL 'STATS'                    PROCEDURE DIVISION
   USING NO-OF-RECS                USING NO-OF-VALUES
         SALES-TABLE                     INPUT-TABLE
         SALES-STATS                     OUTPUT-STATS
         COMPLETION-CODE                 STATUS-CODE.
```

The sample subprogram could be thought of as having the following function: Receive a table of data and compile the minimum, maximum and average values; if no error occurs in the process, indicate so by placing 'O.K.' in a status field, otherwise put spaces in that field. This description makes no reference to sales data or any particular type of data. Thus, generic data-names such as INPUT-TABLE and OUTPUT-STATS are reasonable choices, and serve to emphasize the point that a subprogram may be designed to perform a general-purpose function and could therefore be used (CALLED) by *many* programs needing execution of that same function.

Incidentally, the . . . OCCURS DEPENDING ON could have been used in the table description of either the main program or the subprogram. It was not used here because the SEARCH verb, which takes direct advantage of this feature, was not included in the program.

COBOL STATEMENTS FOR SUBPROGRAMS

There are several specialized COBOL statements for use with respect to subprograms. In this section we consider some of the options associated with the CALL, PROCEDURE DIVISION USING, and EXIT PROGRAM statements. Other specialized statements, such as the CANCEL and USE, are outside the intended scope of this book.

The LINKAGE SECTION is illustrated in the subprogram in Figure 15-3. It is written in the DATA DIVISION following the WORKING-STORAGE SECTION and includes all the data-names that appear in the PROCEDURE DIVISION USING . . . list. Items in that list should appear in the LINKAGE SECTION either as level 01 or 77 items. If an item is a group item, it must be at the 01 level. If it is elementary then it may be at the 77 level, although use of this level is being phased out of the language.

In parallel, the items in the CALL . . . USING . . . list of the calling program must also be at the 01 or 77 level.

Figure 15-4 presents the two general formats for the CALL statement. Given that our objective in this section is to give you an overview of the options associated with this command, rather than a comprehensive coverage, following is an abbreviated description of the options:

Identifier-1. It is possible not to specify the name of the called subprogram as an explicit literal, as the following example illustrates:

```
IF RESPONSE = 'M' MOVE 'MONTHLY' TO SUB-NAME
ELSE MOVE 'QUARTERLY' TO SUB-NAME.
CALL SUB-NAME USING . . .
```

Notice that SUB-NAME is a data-name, not a literal.

BY REFERENCE. This clause may be omitted, in which case it is specified by default, as was done in the sample subprogram in Figure 15-3. BY REFERENCE means to associate subprogram data-names in the LINKAGE

```
format 1:

CALL {identifier-1}
     {literal-1   }

[       {[BY REFERENCE]  identifier-2 . . .}      ]
[ USING {                                   } . . .]
[       {BY CONTENT      identifier-2      }      ]

[ON OVERFLOW imperative-statement-1  [END-CALL]

format 2:

CALL {identifier-1}
     {literal-1   }

[       {[BY REFERENCE]  identifier-2 . . .}      ]
[ USING {                                   } . . .]
[       {BY CONTENT      identifier-2 . . .}      ]

[ON EXCEPTION imperative statement-1]
[NOT ON EXCEPTION imperative-statement-2]
[END-CALL]
```

FIGURE 15-4
GENERAL FORMATS FOR THE CALL STATEMENT.

and PROCEDURE DIVISION USING . . . specifications with the address of the corresponding data-names in the calling program.

BY CONTENT. The CALL statement makes available to the subprogram the content of the data-names listed in the CALL statement. The subprogram then does not make reference to the actual data-name storage locations in the calling program and cannot alter the contents of the data-names in the calling program. In essence, the calling program passes data, not references, to the called program. In Figure 15-2, SALES-STATS and COMPLETION-CODE should definitely *not* be declared USING . . . BY CONTENT SALES-STATS COMPLETION CODE, because the calling program is relying on the data items being altered. However, the other two arguments could have been passed BY CONTENT, since they are used only in the one direction of sending data to the called program. Thus we could say:

```
CALL 'STATS' USING BY CONTENT NO-OF-RECS, SALES-DATA
                   BY REFERENCE SALES-STATS, COMPLETION-CODE.
```

ON OVERFLOW. This conditional specifies execution of imperative-statement-1 if the CALL cannot be executed because of insufficient storage, and the like.

END-CALL. This is a scope terminator that can be used as follows:

```
IF N > ZERO
      CALL 'STATS' USING . . . .
          ON OVERFLOW
             PERFORM ABC
             MOVE A TO Z
      END-CALL
ELSE . . . .
```

The second format of the CALL in Figure 15-4 uses the conditional ON EXCEPTION and NOT ON EXCEPTION as a more flexible alternative to the ON OVERFLOW of format-1. An EXCEPTION indicates an error condition, such as not being able to access the subprogram, insufficient storage, and the like.

The last two specialized COBOL statements for subprograms are the PROCEDURE DIVISION USING and the EXIT PROGRAM statements.

The PROCEDURE DIVISION USING . . . lists the data-names that correspond to the CALL statement and that are described in the LINKAGE SECTION. It is possible to have a subprogram without arguments. For instance, we could have a subprogram that opens a file, generates a report from data in that file, closes the file, and exits. In such a case there would be no need for arguments. Therefore we could have CALL 'SUB' as the complete CALL statement and no USING clause in the PROCEDURE DIVISION header of the 'SUB' subprogram.

The EXIT PROGRAM statement terminates execution of the subprogram and returns execution to the statement following the corresponding CALL statement. In the 1974 ANSI standard version of COBOL, EXIT PROGRAM must be in a paragraph by itself. In the revised version of COBOL, EXIT PROGRAM may be in a paragraph with other statements, just like the STOP RUN statement.

Review

1 In the revised language, the clause in the CALL statement that is used to associate subprogram data-names in the LINKAGE and PROCEDURE DIVISION USING . . . specifications with the address of the corresponding data-names in the calling program is the BY ________________ clause.

REFERENCE

2 In the revised language, the option of the CALL statement that makes available the content of data-names in the calling program, but without the possibility of altering the contents of these storage locations, is the BY ________________ clause.

CONTENT

3 The COBOL statement in the subprogram that lists the data-names that correspond to the CALL statement, and that are described in the LINKAGE SECTION, is the ________________ . . . statement.

PROCEDURE DIVISION USING

4 The statement that terminates execution of the subprogram and returns execution to the statement following the corresponding CALL statement is the ________________ command.

EXIT PROGRAM

NESTED COBOL PROGRAMS

In the revised version of COBOL, it is possible to write programs that contain other programs. Of course, a subprogram is, logically, a program contained within its calling program. However, the new version of COBOL makes available an additional technique by which programs can be contained within, or nested, within another program. One advantage of nested programs is that they not only allow the modularity benefits of subprograms, but they also facilitate data sharing among programs.

In keeping with the scope of this book and the advanced nature of the subject, we will only introduce the topic by means of an example.

Figure 15-5 is a modified version of Figures 15-2 and 15-3 combined. Figure 15-5 includes a run unit consisting of two nested programs. The first program, SALESTATS, is a *containing* program; the second program, STATS, is a *contained* program.

Notice that there is no LINKAGE SECTION and no USING clause in the PROCEDURE DIVISION header of the STATS program. Reviewing the WORKING-STORAGE SECTION of SALESTATS we see the GLOBAL declaration for COMPLETION-CODE, SALES-DATA, and SALES-STATS. The meaning of GLOBAL is that the corresponding data-name is declared in only one program, but may be referenced from any program contained within that program. In our example, STATS is a program contained within SALESTATS. Therefore, STATS can use these GLOBAL data-names without having to describe them in its own DATA DIVISION.

```
IDENTIFICATION DIVISION.
PROGRAM-ID. SALESTATS.
*
ENVIRONMENT DIVISION.
*
CONFIGURATION SECTION.
*
SOURCE-COMPUTER. ABC-480.
OBJECT-COMPUTER. ABC-480.
*
INPUT-OUTPUT SECTION.
FILE-CONTROL.
    SELECT SOURCE-FILE  ASSIGN TO 'SOURCE'.
    SELECT OUTPUT-FILE  ASSIGN TO 'PRINTER'.
*
DATA DIVISION.
*
FILE SECTION.
*
FD  SOURCE-FILE               LABEL RECORDS STANDARD
                              DATA RECORD IS SOURCE-REC.
01  SOURCE-REC.
    02  SALES-AMOUNT          PIC 9(4)V99.
    02  FILLER                PIC X(74).
*
FD  OUTPUT-FILE               LABEL RECORDS OMITTED
                              DATA RECORD IS OUTPUT-REC.
*
01  OUTPUT-REC                PIC X(132).
*
WORKING-STORAGE SECTION.
*
01  END-OF-FILE-TEST          PIC XXX VALUE 'NO '.
    88 END-OF-FILE            VALUE 'YES'.
*
01  COMPLETION-CODE  IS  GLOBAL.
    02     PIC X(4).
*
01  SALES-DATA       IS  GLOBAL.
    02  N-GOOD               PIC 9(3).
    02  N-TOTAL              PIC 9(3).
    02  SALES-TABLE.
        03  SALES            OCCURS 100 TIMES
                             PIC 9(4)V99.
*
01  SALES-STATS      IS  GLOBAL.
    02  MIN                  PIC 9(4)V99.
    02  MAX                  PIC 9(4)V99.
    02  AVG                  PIC 9(4)V99.
```

FIGURE 15-5
SAMPLE NESTED PROGRAMS.

```
 01  WS-OUTPUT-REC.
     02  FILLER                    PIC X(6) VALUE 'MIN = '.
     02  MIN-OUT                   PIC Z(4)9.99.
     02  FILLER                    PIC X(8) VALUE '   MAX = '.
     02  MAX-OUT                   PIC Z(4)9.99.
     02  FILLER                    PIC X(8) VALUE '   AVG = '.
     02  AVG-OUT                   PIC Z(4)9.99.
     02  FILLER                    PIC X(6) VALUE '   N = '.
     02  N-OUT                     PIC ZZ9.
*
*
 PROCEDURE DIVISION.
*
 PROGRAM-SUMMARY.
     OPEN INPUT SOURCE-FILE  OUTPUT OUTPUT-FILE
*
     INITIALIZE SALES-DATA
*
     READ SOURCE-FILE
        AT END
           SET END-OF-FILE TO TRUE
     END-READ
*
     PERFORM STORE-SALES
         UNTIL END-OF-FILE
         OR    N-GOOD = 100
*
     END-PERFORM
*
     IF N-GOOD > ZERO
       THEN
         CALL 'STATS'
*
         IF COMPLETION-CODE = 'O.K.'
           THEN
             NEXT SENTENCE
           ELSE
             MOVE ZEROS TO SALES-STATS
         END-IF
*
       ELSE
         MOVE ZEROS TO SALES-STATS
     END-IF
*
     PERFORM PRINT-STATS
*
     CLOSE SOURCE-FILE OUTPUT-FILE
*
     STOP RUN.
 STORE-SALES.
     ADD 1 TO N-TOTAL
*
     IF SALES-AMOUNT IS NUMERIC
       THEN
         ADD 1 TO N-GOOD
         MOVE SALES-AMOUNT TO SALES (N-GOOD)
       ELSE
*
         STRING 'RECORD ', N-TOTAL,
                ' CONTAINS NONNUMERIC DATA: '
                SOURCE-REC(1:6)
                DELIMITED BY SIZE
           INTO OUTPUT-REC
         WRITE OUTPUT-REC
*
     END-IF
*
     READ SOURCE-FILE
        AT END
           SET END-OF-FILE TO TRUE
     END-READ.
```

FIGURE 15-5 *(Continued)*
SAMPLE NESTED PROGRAMS.

```
*
PRINT-STATS.
    MOVE MIN TO MIN-OUT
    MOVE MAX TO MAX-OUT
    MOVE AVG TO AVG-OUT
    MOVE N-GOOD TO N-OUT
    WRITE OUTPUT-REC FROM WS-OUTPUT-REC.
*
*   The following program is a contained program.
*
IDENTIFICATION DIVISION.
PROGRAM-ID. STATS.
*
DATA DIVISION.
*
WORKING-STORAGE SECTION.
*
01  SALES-TOTAL             PIC 9(5)V99 VALUE ZERO.
01  I                       PIC 999.
PROCEDURE DIVISION.
*
PROGRAM-SUMMARY.
    MOVE 'O.K.' TO COMPLETION-CODE
    MOVE SALES (1) TO MIN, MAX, SALES-TOTAL.
    PERFORM  VARYING I FROM 1 BY 1
                       UNTIL I > N-GOOD
                       OR COMPLETION-CODE = SPACES
      IF MIN > SALES (I)
         THEN
           MOVE SALES (I) TO MIN
         ELSE
           IF MAX < SALES (I)
              THEN
                MOVE SALES (I) TO MAX
              ELSE
                CONTINUE
           END-IF
      END-IF
*
      ADD SALES (I) TO SALES-TOTAL
          ON SIZE ERROR MOVE SPACES TO COMPLETION-CODE
      END-ADD
*
   END-PERFORM
*
   DIVIDE SALES-TOTAL BY N-GOOD GIVING AVG
          ON SIZE ERROR MOVE SPACES TO COMPLETION-CODE.
*
RETURN-TO-CALL.
    EXIT PROGRAM.
*
END PROGRAM STATS.
*
END PROGRAM SALESTATS.
```

FIGURE 15-5 *(Continued)*
SAMPLE NESTED PROGRAMS.

Reviewing the DATA DIVISION of the STATS program, we see that there are two data-names defined. These are *local* (as opposed to global) data-names whose meaning is local to the defining program. For example, if SALES-TOTAL had also been defined in SALESTATS, use of SALES-TOTAL within SALE-STATS would be completely different from the use of SALES-TOTAL within STATS.

Another point to notice is the absence of an ENVIRONMENT DIVISION and the abbreviated version of the DATA DIVISION in the contained program, STATS. It need not have been that way if the specific application were different, but the example illustrates that in a run unit consisting of multiple programs, some divisions may be left out. The IDENTIFICATION DIVISION must always be included, however.

The last two lines in Figure 15-5 demonstrate use of the END PROGRAM header. The scope of each nested program is terminated by such a header. Figure 15-6 further illustrates use of END PROGRAM in a run unit consisting of six nested programs, and provides a more extensive example of nesting programs. The large vertical brackets on the right side of Figure 15-6 are shown to help the reader identify the scope of each program.

The illustration in Figure 15-5 and the discussion above provide a basic idea of the nature and purpose of nested programs. In a way, nested program structures provide something between the use of a PERFORM and the use of a CALL. In essence, the traditional PERFORM construct relies on all data being global, and each object of PERFORM can be thought of as a contained program. The CALL construct, on the other hand, does not allow physical inclusion of nested programs and requires that all data definitions be local.

Nested programming in the revised version of COBOL includes many other features, such as EXTERNAL declarations for files and data-names and COMMON and INITIAL declarations for contained programs. Descriptions of such treatments are included in more advanced coverage of the language features.

```
IDENTIFICTION DIVISION.
PROGRAM-ID. MASTER-PROGRAM.
  :
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM A.
  :
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM B.
  :
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM C.
  :
PROCEDURE DIVISION.
  :
END PROGRAM PROGRAM-C.
END PROGRAM PROGRAM-B.
END PROGRAM PROGRAM-A.
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-Y.
  :
IDENTIFICATION DIVISION.
PROGRAM-ID. PROGRAM-Y.
  :
PROCEDURE DIVISION.
  :
END PROGRAM PROGRAM-Y.
END PROGRAM PROGRAM-X.
END PROGRAM MASTER-PROGRAM.
```

FIGURE 15-6
SAMPLE OUTLINE OF MULTIPLE-NESTED PROGRAMS.

Review

1 In addition to the use of subprograms, the new version of COBOL makes it possible to have programs that are contained *within* another program through the process of ________________.

nesting

2 In the context of nesting, the main program is called the *containing* program while the nested programs are called the ________________ programs.

contained

3 When nesting is used, detailed DATA DIVISION specifications [need to / need not] be included in the contained programs.

need not

4 The declaration in the WORKING-STORAGE SECTION of the containing program that makes it possible to reference the data-names in the contained programs is the ________________ declaration.

GLOBAL

5 The DATA DIVISION of a contained program can include data-names that are not defined in the containing program. As opposed to global data-names, such data-names are called ________________ data-names.

local

6 When nested programs are used, some divisions of the COBOL program can be omitted in a contained program. However, the one division that must be included is the ________________ DIVISION.

IDENTIFICATION

EXERCISES

15.1 Discuss three main reasons for using externally compiled subprograms rather than using PERFORM structures within a large program.

15.2 Consider the following:

```
Main Program
CALL 'CHR-PROC' USING X, 'SAMPLE'.
.
.

Subprogram
LINKAGE SECTION.
01   A   PIC   X(5).
01   B   PIC   X(3).
PROCEDURE DIVISION USING B, A.
.
.
.
MOVE SPACES TO A.
```

What will happen as a result of the MOVE statement in the subprogram?

15.3 Employees pay FICA-RATE percent of their first FICA-LIMIT annual earnings as F.I.C.A. tax. We wish to write a subprogram that will compute

the F.I.C.A. tax for each employee. Outline a skeleton program structure for a main program and the subprogram. The main program CALLs the subprogram to compute the F.I.C.A. tax. Be sure to include argument definition and PROCEDURE DIVISION statements for both the main program and the subprogram.

15.4 A subprogram is to be written which can perform the following functions:

Open the printer file

Close the printer file

Write a record, leaving a specified number of blank lines before printing

Design such a subprogram by responding to the following requirements:

a Write the LINKAGE SECTION of the subprogram (named PRINTER).

b Write the entire PROCEDURE DIVISION of the subprogram.

c Write sample CALL statements for the main program.

15.5 Your task is to prepare a program to check for errors in transaction data pertaining to inventory. The source data items have the following format:

COLUMN	FIELD-NAME
1–5	ITEM-NUMBER
6–20	ITEM-NAME
21	ITEM-CODE
22–26	QUANTITY

The basic checking procedure is concerned with the value of ITEM-CODE, because this signifies the type of record. The value 1 means that the QUANTITY field contains the previous balance for the item specified by ITEM-NUMBER; the value 2 indicates the receipt of goods; and 3 indicates the issue of goods from inventory. Any other code is an error. We assume that the records are sorted so that all records of the same item number are grouped together and the record with a 1 in column 21 leads the group. We recognize four types of error conditions:

Duplicate balance record: This condition arises whenever more than one record in a group of the same item number has a code of 1.

Misplaced balance record: A record with a code of 1 exists, but it is not the first in the group.

Balance record missing: The first record in a group is not a code 1 record.

Incorrect code: A code other than 1, 2, or 3 exists.

On detection of a record meeting one of these conditions, the record is printed with the corresponding explanatory error message so that it can be corrected.

In addition to the error messages, we also desire a summary of totals. Figure 15-7 illustrates a set of sample input records and the resulting sample output.

We proceed to define the function of the main program and the subprogram as follows:

Main program: This reads each transaction and gives each record to the subprogram to check for data validity. If execution of the subprogram indicates that the data are not valid, the record is printed along with an

```
Sample Input

12345TEST-ITEM-1     100100
12345TEST-ITEM-1     200100
12345TEST-ITEM-1     300200
23456TEST-ITEM-2     300010
34567TEST-ITEM-3     100020
34567TEST-ITEM-3     200100
34567TEST-ITEM-3     300050
45678TEST-ITEM-4     200100
45678TEST-ITEM-4     100100
45678TEST-ITEM-4     300100
45678TEST-ITEM-4     100200
56789TEST-ITEM-5     100100
56789TEST-ITEM-5     40050
56789TEST-ITEM-5     300020
67890TEST-ITEM-6     100300
67890TEST-ITEM-6     300100
78901TEST-ITEM-7     100400
78901TEST-ITEM-7     100300
89012TEST-ITEM-8     100200

Sample Output

23456TEST-ITEM-2     300010    BALANCE CARD MISSING
45678TEST-ITEM-4     200100    BALANCE CARD MISSING
45678TEST-ITEM-4     100100    MISPLACED BALANCE CARD
45678TEST-ITEM-4     300100    BALANCE CARD MISSING
45678TEST-ITEM-4     100200    MISPLACED BALANCE CARD
56789TEST-ITEM-5     40050     INCORRECT CODE
78901TEST-ITEM-7     100300    DUPLICATE BALANCE CARD

VALID RECORDS =                          012

INVALID RECORDS =                        007

BALANCE TOTAL =                         1120

RECEIPTS TOTAL =                         200

ISSUES TOTAL =                           370
```

FIGURE 15-7
SAMPLE INPUT AND OUTPUT FOR THE INVENTORY EXAMPLE.

error message. If execution of the subprogram indicates that the data are valid, then we accumulate the proper totals. We then proceed to read another record. When all the records have been read, we print a summary of the accumulated totals and terminate the program.

Subprogram: The subprogram receives a transaction record from the main program. It checks for errors. If an error is found, an appropriate error message is supplied to the main program. If no error is found, a blank error message is supplied to the main program. Then the subprogram terminates.

The four error messages are:

DUPLICATE BALANCE CARD
MISPLACED BALANCE CARD
BALANCE CARD MISSING
INCORRECT CODE

Figure 15-8 is a *decision table* which facilitates complete enumeration of the possibilities.

	POSSIBILITY							
CONDITION	1	2	3	4	5	6	7	8
PREVIOUS-ITEM = CURRENT-ITEM	X	X	X					
PREVIOUS-ITEM NOT = CURRENT-ITEM				X	X	X	X	X
KODE = 1	X			X	X			
KODE = 2 OR 3		X				X		X
KODE NOT = 1 OR 2 OR 3			X				X	
BALANCE-CODE = 1				X		X		
BALANCE-CODE = 2					X			X
ACTION	1	2	3	4	5	6	7	8
BLANK ERROR MESSAGE	X					X		
DUPLICATE BALANCE CARD				X				
MISPLACED BALANCE CARD					X			
BALANCE CARD MISSING		X	X					X
INCORRECT CODE							X	

FIGURE 15-8
DECISION TABLE FOR THE INVENTORY EXAMPLE.

A COBOL reserved words

ACCEPT
ACCESS
ADD
ADVANCING
AFTER
ALL
ALPHABET
ALPHABETIC
ALPHABETIC-
  LOWER
ALPHABETIC-UPPER
ALPHANUMERIC
ALPHANUMERIC-
  EDITED
ALSO
ALTER
ALTERNATE
AND
ANY
ARE
AREA
AREAS
ASCENDING
ASSIGN
AT
AUTHOR

BEFORE

BINARY
BLANK
BLOCK
BOTTOM
BY

CALL
CANCEL
CD
CF
CH
CHARACTER
CHARACTERS
CLASS
CLOSE
CODE
CODE-SET
COLLATING
COLUMN
COMMA
COMMON
COMMUNICATION
COMP
COMPUTATIONAL
COMPUTE
CONFIGURATION
CONTAINS
CONTENT

CONTINUE
CONTROL
CONTROLS
CONVERTING
COPY
CORR
CORRESPONDING
COUNT
CURRENCY

DATA
DATE
DATE-COMPILED
DATE-WRITTEN
DAY
DAY-OF-WEEK
DE
DEBUG-CONTENTS
DEBUG-ITEM
DEBUG-LINE
DEBUG-NAME
DEBUG-SUB-1
DEBUG-SUB-2
DEBUG-SUB-3
DEBUGGING
DECIMAL-POINT
DECLARATIVES

DELETE
DELIMITED
DELIMITER
DEPENDING
DESCENDING
DESTINATION
DETAIL
DISABLE
DISPLAY
DIVIDE
DIVISION
DOWN
DUPLICATES
DYNAMIC

EGI
ELSE
EMI
ENABLE
END
END-ADD
END-CALL
END-COMPUTE
END-DELETE
END-DIVIDE
END-EVALUATE
END-IF
END-MULTIPLY

END-OF-PAGE
END-PERFORM
END-READ
END-RECEIVE
END-RETURN
END-REWRITE
END-SEARCH
END-START
END-STRING
END-SUBTRACT
END-UNSTRING
END-WRITE
ENVIRONMENT
EOP
EQUAL
ERROR
ESI
EVALUATE
EXCEPTION
EXIT
EXTEND
EXTERNAL

FALSE
FD
FILE
FILE-CONTROL
FILLER
FINAL
FIRST
FOOTING
FOR
FROM

GENERATE
GIVING
GLOBAL
GO
GREATER
GROUP

HEADING
HIGH-VALUE
HIGH-VALUES

I-O
I-O-CONTROL
IDENTIFICATION
IF
IN
INDEX
INDEXED
INDICATE
INITIAL
INITIALIZE
INITIATE
INPUT
INPUT-OUTPUT
INSPECT
INSTALLATION
INTO
INVALID
IS

JUST
JUSTIFIED

KEY

LABEL
LAST
LEADING
LEFT
LENGTH
LESS
LIMIT
LIMITS
LINAGE
LINAGE-COUNTER
LINE
LINE-COUNTER
LINES
LINKAGE
LOCK
LOW-VALUE
LOW-VALUES

MERGE
MESSAGE
MODE
MOVE
MULTIPLE
MULTIPLY

NATIVE
NEGATIVE
NEXT
NO
NOT
NUMBER
NUMERIC
NUMERIC-EDITED

OBJECT-COMPUTER
OCCURS
OF
OFF
OMITTED
ON
OPEN
OPTIONAL
OR
ORDER
ORGANIZATION
OTHER
OUTPUT
OVERFLOW

PACKED-DECIMAL
PADDING
PAGE
PAGE-COUNTER
PERFORM
PF
PH
PIC
PICTURE
PLUS
POINTER
POSITION
POSITIVE
PRINTING
PROCEDURE
PROCEDURES
PROCEED
PROGRAM
PROGRAM-ID
PURGE

QUEUE
QUOTE
QUOTES

RANDOM
RD
READ
RECEIVE
RECORD
RECORDS
REDEFINES
REEL
REFERENCE
REFERENCES
RELATIVE
RELEASE
REMAINDER
REMOVAL
RENAMES
REPLACE
REPLACING
REPORT
REPORTING
REPORTS
RESERVE
RESET
RETURN
REWIND
REWRITE
RF
RH
RIGHT
ROUNDED
RUN

SAME
SD
SEARCH
SECTION
SECURITY
SEGMENT
SEGMENT-LIMIT
SELECT
SEND
SENTENCE
SEPARATE
SEQUENCE
SEQUENTIAL
SET
SIGN
SIZE
SORT
SORT-MERGE
SOURCE
SOURCE-COMPUTER
SPACE
SPACES
SPECIAL-NAMES
STANDARD
STANDARD-1
STANDARD-2
START
STATUS
STOP
STRING
SUB-QUEUE-1
SUB-QUEUE-2
SUB-QUEUE-3
SUBTRACT
SUM
SUPPRESS
SYMBOLIC
SYNC
SYNCHRONIZED

TABLE
TALLYING
TAPE
TERMINAL
TERMINATE
TEST
TEXT

THAN
THEN
THROUGH
THRU
TIME
TIMES
TO
TOP
TRAILING
TRUE
TYPE

UNIT
UNSTRING
UNTIL
UP
UPON
USAGE
USE
USING

VALUE
VALUES

VARYING

WHEN
WITH
WORKING-STORAGE
WRITE

ZERO
ZEROES
ZEROS

+
−
*
/
**
>
<
=
>=
<=

B COMPLETE LANGUAGE FORMATS FOR THE REVISED VERSION OF COBOL

This appendix contains the composite language skeleton of the revised version of the American National Standard COBOL. It is intended to display complete and syntactically correct formats.

The leftmost margin on pages 387 through 388 and pages 391 through 398 is equivalent to margin A in a COBOL source program. The first indentation after the leftmost margin is equivalent to margin B in a COBOL source program.

On pages 399 through 411 the leftmost margin indicates the beginning of the format for a new COBOL verb. The first indentation after the leftmost margin indicates continuation of the format of the COBOL verb. The appearance of the italic letter *S, R, I,* or *W* to the left of the format for the verbs CLOSE, OPEN, READ, and WRITE indicates the Sequential I-O module, Relative I-O module, Indexed I-O module, or Report Writer module in which that general format is used.

The following are the formats presented on pages 387 through 415:

GENERAL FORMAT FOR IDENTIFICATION DIVISION

```
IDENTIFICATION DIVISION.

PROGRAM-ID. program-name [IS {|COMMON |} PROGRAM]
                              {|INITIAL|}

[AUTHOR. [comment-entry] ... ]

[INSTALLATION. [comment-entry] ... ]

[DATE-WRITTEN. [comment-entry] ... ]

[DATE-COMPILED. [comment-entry] ... ]

[SECURITY. [comment-entry] ... ]
```

GENERAL FORMAT FOR ENVIRONMENT DIVISION

```
[ENVIRONMENT DIVISION.

[CONFIGURATION SECTION.

[SOURCE-COMPUTER.  [computer-name  [WITH DEBUGGING MODE].]]

[OBJECT-COMPUTER.  [computer-name

    [PROGRAM COLLATING SEQUENCE IS alphabet-name-1]

    [SEGMENT-LIMIT IS segment-number].]]

[SPECIAL-NAMES.  [[implementor-name-1

    { IS mnemonic-name-1  [ON STATUS IS condition-name-1  [OFF STATUS IS condition-name-2]] }
    { IS mnemonic-name-2  [OFF STATUS IS condition-name-2  [ON STATUS IS condition-name-1]] }
    { ON STATUS IS condition-name-1  [OFF STATUS IS condition-name-2]                       } ] ...
    { OFF STATUS IS condition-name-2  [ON STATUS IS condition-name-1]                       }

  [ALPHABET alphabet-name-1 IS

    { STANDARD-1                                                         }
    { STANDARD-2                                                         }
    { NATIVE                                                             }
    { implementor-name-2                                                 } ] ...
    {   {              [ {THROUGH}  literal-2        ] }                 }
    {   { literal-1    [ {THRU   }                   ] } ...             }
    {   {              [ {ALSO literal-3} ...         ] }                }

  [SYMBOLIC CHARACTERS {{symbolic-character-1} ... {IS } {integer-1} ... } ...
                                                   {ARE}
       [IN alphabet-name-2]] ...

  [CLASS class-name IS {literal-4  [{THROUGH} literal-5]} ... ] ...
                                    {THRU   }

  [CURRENCY SIGN IS literal-6]

  [DECIMAL-POINT IS COMMA].]]]

  [INPUT-OUTPUT SECTION.

   FILE-CONTROL.

      {file-control-entry} ...

  [I-O-CONTROL.

     [[SAME [RECORD     ] AREA FOR file-name-1  {file-name-2} ... ] ...
            [SORT       ]
            [SORT-MERGE ]

     [MULTIPLE FILE TAPE CONTAINS

        {file-name-3  [POSITION IS integer-1] } ... ] ...  .]]]]
```

GENERAL FORMAT FOR FILE CONTROL ENTRY

SEQUENTIAL FILE:

```
SELECT [OPTIONAL] file-name-1

    ASSIGN TO {implementor-name-1} ...
              {literal-1         }

    [RESERVE integer-1 [AREA ]]
                       [AREAS]

    [[ORGANIZATION IS] SEQUENTIAL]

    [PADDING CHARACTER IS {data-name-1}]
                          {literal-2  }

    [RECORD DELIMITER IS {STANDARD-1        }]
                         {implementor-name-2}

    [ACCESS MODE IS SEQUENTIAL]

    [FILE STATUS IS data-name-2].
```

RELATIVE FILE:

```
SELECT [OPTIONAL] file-name-1

    ASSIGN TO {implementor-name-1} ...
              {literal-1         }

    [RESERVE integer-1 [AREA ]]
                       [AREAS]

    [ORGANIZATION IS] RELATIVE

    [                {SEQUENTIAL  [RELATIVE KEY IS data-name-1]}]
    [ACCESS MODE IS  {                                         }]
    [                {{RANDOM }                                }]
    [                {{DYNAMIC}   RELATIVE KEY IS data-name-1  }]

    [FILE STATUS IS data-name-2].
```

INDEXED FILE:

```
SELECT [OPTIONAL] file-name-1

    ASSIGN TO {implementor-name-1} ...
              {literal-1         }

    [RESERVE integer-1 [AREA ]]
                       [AREAS]

    [ORGANIZATION IS] INDEXED

    [                {SEQUENTIAL}]
    [ACCESS MODE IS  {RANDOM    }]
    [                {DYNAMIC   }]

    RECORD KEY IS data-name-1

    [ALTERNATE RECORD KEY IS data-name-2 [WITH DUPLICATES]] ...

    [FILE STATUS IS data-name-3].
```

SORT OR MERGE FILE:

```
SELECT file-name-1 ASSIGN TO {implementor-name-1} ... .
                             {literal-1         }
```

GENERAL FORMAT FOR FILE CONTROL ENTRY

```
REPORT FILE:

SELECT [OPTIONAL] file-name-1

    ASSIGN TO {implementor-name-1} ...
              {literal-1         }

    [RESERVE integer-1 [AREA ]]
                       [AREAS]

    [[ORGANIZATION IS] SEQUENTIAL]

    [PADDING CHARACTER IS {data-name-1}]
                          {literal-1  }

    [RECORD DELIMITER IS {STANDARD-1        }]
                         {implementor-name-2}

    [ACCESS MODE IS SEQUENTIAL]

    [FILE STATUS IS data-name-2].
```

GENERAL FORMAT FOR DATA DIVISION

```
[DATA DIVISION.

[FILE SECTION.

[file-description-entry

{record-description-entry} ... ] ...

[sort-merge-file-description-entry

{record-description-entry} ... ] ...

[report-file-description-entry] ... ]

[WORKING-STORAGE SECTION.

[77-level-description-entry ] ...
[record-description-entry   ]

[LINKAGE SECTION.

[77-level-description-entry ] ...
[record-description-entry   ]

[COMMUNICATION SECTION.

[communication-description-entry

[record-description-entry] ... ] ... ]

[REPORT SECTION.

[report-description-entry

{report-group-description-entry} ... ] ... ]]
```

GENERAL FORMAT FOR FILE DESCRIPTION ENTRY

SEQUENTIAL FILE:

```
FD file-name-1

   [IS EXTERNAL]

   [IS GLOBAL]

   [BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS   }]
                                            {CHARACTERS}

   [       {CONTAINS integer-3 CHARACTERS                                          }]
   [RECORD {IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS]       }]
   [       {     [DEPENDING ON data-name-1]                                        }]
   [       {CONTAINS integer-6 TO integer-7 CHARACTERS                             }]

   [LABEL {RECORD IS   } {STANDARD}]
   [      {RECORDS ARE } {OMITTED }]

   [VALUE OF {implementor-name-1 IS {data-name-2}} ... ]
   [                                {literal-1  }      ]

   [DATA {RECORD IS  } {data-name-3} ... ]
   [     {RECORDS ARE}                   ]

   [LINAGE IS {data-name-4} LINES [WITH FOOTING AT {data-name-5}]
   [          {integer-8  }       [                {integer-9  }]

        [LINES AT TOP {data-name-6}] [LINES AT BOTTOM {data-name-7}]]
        [             {integer-10 }] [                {integer-11 }]]

   [CODE-SET IS alphabet-name-1].
```

GENERAL FORMAT FOR FILE DESCRIPTION ENTRY

RELATIVE FILE:

```
FD  file-name-1

    [IS EXTERNAL]

    [IS GLOBAL]

    [BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS   }]
                                             {CHARACTERS}

    [       {CONTAINS integer-3 CHARACTERS                                         }]
    [RECORD {IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS]       }]
    [       {     [DEPENDING ON data-name-1]                                       }]
    [       {CONTAINS integer-6 TO integer-7 CHARACTERS                            }]

    [LABEL {RECORD IS  } {STANDARD}]
    [      {RECORDS ARE} {OMITTED }]

    [VALUE OF {implementor-name-1 IS {data-name-2}} ...]
    [                                {literal-1  }     ]

    [DATA {RECORD IS  } {data-name-3} ...]
    [     {RECORDS ARE}                  ]
```

INDEXED FILE:

```
FD  file-name-1

    [IS EXTERNAL]

    [IS GLOBAL]

    [BLOCK CONTAINS [integer-1 TO] integer-2 {RECORDS   }]
                                             {CHARACTERS}

    [       {CONTAINS integer-3 CHARACTERS                                         }]
    [RECORD {IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS]       }]
    [       {     [DEPENDING ON data-name-1]                                       }]
    [       {CONTAINS integer-6 TO integer-7 CHARACTERS                            }]

    [LABEL {RECORD IS  } {STANDARD}]
    [      {RECORDS ARE} {OMITTED }]

    [VALUE OF {implementor-name-1 IS {data-name-2}} ...]
    [                                {literal-1  }     ]

    [DATA {RECORD IS  } {data-name-3} ...] .
    [     {RECORDS ARE}                  ]
```

GENERAL FORMAT FOR FILE DESCRIPTION ENTRY

SORT-MERGE FILE:

```
SD  file-name-1

    [        {CONTAINS integer-1 CHARACTERS                                          }]
    [        {IS VARYING IN SIZE [[FROM integer-2] [TO integer-3] CHARACTERS]         }]
    [RECORD  {      [DEPENDING ON data-name-1]                                        }]
    [        {CONTAINS integer-4 TO integer-5 CHARACTERS                              }]

    [DATA {RECORD IS  } {data-name-2} ...]
    [     {RECORDS ARE}                  ]
```

REPORT FILE:

```
FD  file-name-1

    [IS EXTERNAL]

    [IS GLOBAL]

    [BLOCK CONTAINS  [integer-1 TO]  integer-2  {RECORDS   }]
    [                                           {CHARACTERS}]

    [        {CONTAINS integer-3 CHARACTERS                                          }]
    [        {IS VARYING IN SIZE [[FROM integer-4] [TO integer-5] CHARACTERS]         }]
    [RECORD  {      [DEPENDING ON data-name-1]                                        }]
    [        {CONTAINS integer-6 TO integer-7 CHARACTERS                              }]

    [LABEL  {RECORD IS  }  {STANDARD}]
    [       {RECORDS ARE}  {OMITTED }]

    [VALUE OF  {implementor-name-1 IS  {data-name-2}} ...]
    [                                  {literal-1  }     ]

    [CODE-SET IS alphabet-name-1]

    {REPORT IS  }  {report-name-1} ...
    {REPORTS ARE}
```

GENERAL FORMAT FOR DATA DESCRIPTION ENTRY

FORMAT 1:

```
level-number [data-name-1]
             [FILLER     ]

     [REDEFINES data-name-2]

     [IS EXTERNAL]

     [IS GLOBAL]

     [{PICTURE} IS character-string]
     [{PIC    }                    ]

     [              {BINARY         }]
     [              {COMPUTATIONAL  }]
     [[USAGE IS]    {COMP           }]
     [              {DISPLAY        }]
     [              {INDEX          }]
     [              {PACKED-DECIMAL }]

     [[SIGN IS] {LEADING } [SEPARATE CHARACTER]]
     [          {TRAILING}                     ]

     [OCCURS integer-2 TIMES
          [{ASCENDING } KEY IS {data-name-3} ... ] ...
          [{DESCENDING}                          ]
          [INDEXED BY {index-name-1} ... ]
      OCCURS integer-1 TO integer-2 TIMES DEPENDING ON data-name-4
          [{ASCENDING } KEY IS {data-name-3} ... ] ...
          [{DESCENDING}                          ]
          [INDEXED BY {index-name-1} ... ]                         ]

     [{SYNCHRONIZED} [LEFT ]]
     [{SYNC        } [RIGHT]]

     [{JUSTIFIED} RIGHT]
     [{JUST     }      ]

     [BLANK WHEN ZERO]

     [VALUE IS literal-1].
```

FORMAT 2:

```
66 data-name-1 RENAMES data-name-2 [{THROUGH} data-name-3] .
                                   [{THRU   }            ]
```

FORMAT 3:

```
88 condition-name-1 {VALUE IS  } {literal-1 [{THROUGH} literal-2]} ... .
                    {VALUES ARE}            [{THRU   }          ]
```

GENERAL FORMAT FOR COMMUNICATION DESCRIPTION ENTRY

FORMAT 1:

```
CD  cd-name-1

                               [[SYMBOLIC QUEUE IS data-name-1]
                                    [SYMBOLIC SUB-QUEUE-1 IS data-name-2]
                                    [SYMBOLIC SUB-QUEUE-2 IS data-name-3]
                                    [SYMBOLIC SUB-QUEUE-3 IS data-name-4]
                                    [MESSAGE DATE IS data-name-5]
                                    [MESSAGE TIME IS data-name-6]
                                    [SYMBOLIC SOURCE IS data-name-7]
  FOR [INITIAL] INPUT               [TEXT LENGTH IS data-name-8]
                                    [END KEY IS data-name-9]
                                    [STATUS KEY IS data-name-10]
                                    [MESSAGE COUNT IS data-name-11]]
                               [data-name-1, data-name-2, data-name-3,
                                    data-name-4, data-name-5, data-name-6,
                                    data-name-7, data-name-8, data-name-9,
                                    data-name-10, data-name-11]
```

FORMAT 2:

```
CD  cd-name-1 FOR OUTPUT
    [DESTINATION COUNT IS data-name-1]
    [TEXT LENGTH IS data-name-2]
    [STATUS KEY IS data-name-3]
    [DESTINATION TABLE OCCURS integer-1 TIMES
        [INDEXED BY {index-name-1} ... ]]
    [ERROR KEY IS data-name-4]
    [SYMBOLIC DESTINATION IS data-name-5].
```

FORMAT 3:

```
CD  cd-name-1

                           [[MESSAGE DATE IS data-name-1]
                                [MESSAGE TIME IS data-name-2]
                                [SYMBOLIC TERMINAL IS data-name-3]
                                [TEXT LENGTH IS data-name-4]
  FOR [INITIAL] I-O             [END KEY IS data-name-5]
                                [STATUS KEY IS data-name-6]]
                           [data-name-1, data-name-2, data-name-3,
                                data-name-4, data-name-5, data-name-6]
```

GENERAL FORMAT FOR REPORT DESCRIPTION ENTRY

```
RD report-name-1

   [IS GLOBAL]

   [CODE literal-1]

   [{CONTROL IS  } {{data-name-1} ...             }]
   [{CONTROLS ARE} {FINAL [data-name-1] ...       }]

   [PAGE [LIMIT IS  ] integer-1 [LINE ] [HEADING integer-2]
   [     [LIMITS ARE]           [LINES]

         [FIRST DETAIL integer-3]  [LAST DETAIL integer-4]

         [FOOTING integer-5]].
```

GENERAL FORMAT FOR REPORT GROUP DESCRIPTION ENTRY

FORMAT 1:

```
01  [data-name-1]

   [LINE NUMBER IS {integer-1 [ON NEXT PAGE]}]
   [               {PLUS integer-2          }]

   [                {integer-3     }]
   [NEXT GROUP IS   {PLUS integer-4}]
   [                {NEXT PAGE     }]

             {{REPORT HEADING}                               }
             {{RH            }                               }
             {                                               }
             {{PAGE HEADING}                                 }
             {{PH          }                                 }
             {                                               }
             {{CONTROL HEADING} {data-name-2}                }
             {{CH             } {FINAL      }                }
             {                                               }
   TYPE IS   {{DETAIL}                                       }
             {{DE    }                                       }
             {                                               }
             {{CONTROL FOOTING} {data-name-3}                }
             {{CF             } {FINAL      }                }
             {                                               }
             {{PAGE FOOTING}                                 }
             {{PF          }                                 }
             {                                               }
             {{REPORT FOOTING}                               }
             {{RF            }                               }

   [[USAGE IS] DISPLAY].
```

GENERAL FORMAT FOR REPORT GROUP DESCRIPTION ENTRY

FORMAT 2:

```
level-number [data-name-1]

    [LINE NUMBER IS {integer-1 [ON NEXT PAGE]}]
                    {PLUS integer-2          }

    [[USAGE IS] DISPLAY].
```

FORMAT 3:

```
level-number [data-name-1]

    {PICTURE} IS character-string
    {PIC    }

    [[USAGE IS] DISPLAY]

    [[SIGN IS] {LEADING } SEPARATE CHARACTER]
               {TRAILING}

    [{JUSTIFIED} RIGHT]
    [{JUST     }      ]

    [BLANK WHEN ZERO]

    [LINE NUMBER IS {integer-1 [ON NEXT PAGE]}]
                    {PLUS integer-2          }

    [COLUMN NUMBER IS integer-3]

    {SOURCE IS identifier-1                                            }
    {                                                                  }
    {VALUE IS literal-1                                                }
    {                                                                  }
    {{SUM {identifier-2} ...  [UPON {data-name-2} ... ] } ...          }
    {                                                                  }
    {      [RESET ON {data-name-3}]                                    }
    {                {FINAL      }                                     }

    [GROUP INDICATE].
```

GENERAL FORMAT FOR PROCEDURE DIVISION

FORMAT 1:

```
[PROCEDURE DIVISION  [USING  {data-name-1} ... ].

[DECLARATIVES.

{section-name SECTION [segment-number].

     USE statement.

[paragraph-name.

     [sentence] ... ] ... } ...

 END DECLARATIVES.]

{section-name SECTION [segment-number].

[paragraph-name.

     [sentence] ... ] ... } ... ]
```

FORMAT 2:

```
[PROCEDURE DIVISION  [USING  {data-name-1} ... ].

{paragraph-name.

     [sentence] ... } ... ]
```

GENERAL FORMAT FOR COBOL VERBS

```
ACCEPT identifier-1 [FROM mnemonic-name-1]

                            { DATE        }
ACCEPT identifier-2 FROM    { DAY         }
                            { DAY-OF-WEEK }
                            { TIME        }

ACCEPT cd-name-1 MESSAGE COUNT

ADD {identifier-1} ...  TO {identifier-2 [ROUNDED]} ...
    {literal-1   }

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-ADD]

ADD {identifier-1} ...  TO {identifier-2}
    {literal-1   }         {literal-2   }

    GIVING {identifier-3 [ROUNDED]} ...

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-ADD]

ADD {CORRESPONDING} identifier-1 TO identifier-2 [ROUNDED]
    {CORR         }

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-ADD]

ALTER {procedure-name-1 TO [PROCEED TO] procedure-name-2} ...

CALL {identifier-1} [USING {[BY REFERENCE] {identifier-2} ...} ...]
     {literal-1   }        {BY CONTENT {identifier-2} ...     }

    [ON OVERFLOW imperative-statement-1 [END-CALL]]
```

GENERAL FORMAT FOR COBOL VERBS

```
CALL {identifier-1} [USING {[BY REFERENCE] {identifier-2} ...} ...]
     {literal-1   }         {BY CONTENT {identifier-2} ...   }

    [ON EXCEPTION imperative-statement-1]

    [NOT ON EXCEPTION imperative-statement-2]

    [END-CALL]

CANCEL {identifier-1} ...
       {literal-1   }

                          [{REEL} [FOR REMOVAL]]
SW CLOSE {file-name-1     [{UNIT}              ]} ...
                          [                    ]
                          [WITH {NO REWIND}    ]
                          [     {LOCK     }    ]

RI CLOSE {file-name-1 [WITH LOCK]} ...

COMPUTE {identifier-1 [ROUNDED]} ... = arithmetic-expression-1

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-COMPUTE]

CONTINUE

DELETE file-name-1 RECORD

    [INVALID KEY imperative-statement-1]

    [NOT INVALID KEY imperative-statement-2]

    [END-DELETE]

        {INPUT [TERMINAL]}
DISABLE {I-O TERMINAL    } cd-name-1
        {OUTPUT          }
```

GENERAL FORMAT FOR COBOL VERBS

```
DISPLAY {identifier-1} ... [UPON mnemonic-name-1] [WITH NO ADVANCING]
        {literal-1   }

DIVIDE {identifier-1} INTO {identifier-2 [ROUNDED]} ...
       {literal-1   }

   [ON SIZE ERROR imperative-statement-1]

   [NOT ON SIZE ERROR imperative-statement-2]

   [END-DIVIDE]

DIVIDE {identifier-1} INTO {identifier-2}
       {literal-1   }      {literal-2   }

   GIVING {identifier-3 [ROUNDED]} ...

   [ON SIZE ERROR imperative-statement-1]

   [NOT ON SIZE ERROR imperative-statement-2]

   [END-DIVIDE]

DIVIDE {identifier-1} BY {identifier-2}
       {literal-1   }    {literal-2   }

   GIVING {identifier-3 [ROUNDED]} ...

   [ON SIZE ERROR imperative-statement-1]

   [NOT ON SIZE ERROR imperative-statement-2]

   [END-DIVIDE]

DIVIDE {identifier-1} INTO {identifier-2} GIVING identifier-3 [ROUNDED]
       {literal-1   }      {literal-2   }

   REMAINDER identifier-4

   [ON SIZE ERROR imperative-statement-1]

   [NOT ON SIZE ERROR imperative-statement-2]

   [END-DIVIDE]
```

GENERAL FORMAT FOR COBOL VERBS

```
DIVIDE {identifier-1} BY {identifier-2} GIVING identifier-3 [ROUNDED]
       {literal-1   }    {literal-2   }

    REMAINDER identifier-4

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-DIVIDE]


       {INPUT [TERMINAL]}
ENABLE {I-O TERMINAL    }  cd-name-1
       {OUTPUT          }


         {identifier-1}  [      {identifier-2}]
         {literal-1   }  [      {literal-2   }]
EVALUATE {expression-1}  [ALSO  {expression-2}] ...
         {TRUE        }  [      {TRUE        }]
         {FALSE       }  [      {FALSE       }]

  {{WHEN

    {ANY
    {condition-1
    {TRUE
    {FALSE
    {       {identifier-3              }  [{THROUGH} {identifier-4              }]}}
    {[NOT]  {literal-3                 }  [{THRU   } {literal-4                 }]}}
    {       {arithmetic-expression-1   }  [          {arithmetic-expression-2   }]}}

  [ALSO

    {ANY
    {condition-2
    {TRUE
    {FALSE
    {       {identifier-5              }  [{THROUGH} {identifier-6              }]}} ] ...} ...
    {[NOT]  {literal-5                 }  [{THRU   } {literal-6                 }]}}
    {       {arithmetic-expression-3   }  [          {arithmetic-expression-4   }]}}

   imperative-statement-1} ...

  [WHEN OTHER imperative-statement-2]

  [END-EVALUATE]


EXIT


EXIT PROGRAM
```

GENERAL FORMAT FOR COBOL VERBS

```
GENERATE {data-name-1  }
         {report-name-1}

GO TO [procedure-name-1]

GO TO {procedure-name-1} ... DEPENDING ON identifier-1

IF condition-1 THEN {{statement-1} ...} {ELSE {statement-2} ... [END-IF]}
                    {NEXT SENTENCE    } {ELSE NEXT SENTENCE              }
                                        {END-IF                          }

INITIALIZE {identifier-1} ...
    [           {{ALPHABETIC          }                              }     ]
    [           {{ALPHANUMERIC        }                              }     ]
    [ REPLACING {{NUMERIC             } DATA BY {identifier-2}       } ... ]
    [           {{ALPHANUMERIC-EDITED }         {literal-1   }       }     ]
    [           {{NUMERIC-EDITED      }                              }     ]

INITIATE {report-name-1} ...

INSPECT identifier-1 TALLYING
  {                      { CHARACTERS [{BEFORE} INITIAL {identifier-4}] ...                                                   }     }
  {                      {            [{AFTER }         {literal-2   }]                                                       }     }
  {identifier-2 FOR      {                                                                                                    } ... } ...
  {                      { {ALL    } {{identifier-3} [{BEFORE} INITIAL {identifier-4}] ...} ...                               }     }
  {                      { {LEADING} {{literal-1   } [{AFTER } INITIAL {literal-2   }]    }                                   }     }

INSPECT identifier-1 REPLACING
  { CHARACTERS BY {identifier-5} [{BEFORE} INITIAL {identifier-4}] ...                                          }
  {               {literal-3   } [{AFTER }         {literal-2   }]                                              }
  {                                                                                                             } ...
  { {ALL    } {{identifier-3} BY {identifier-5} [{BEFORE} INITIAL {identifier-4}] ...} ...                      }
  { {LEADING} {{literal-1   }    {literal-3   } [{AFTER }         {literal-2   }]    }                          }
  { {FIRST  }                                                                                                   }
```

GENERAL FORMAT FOR COBOL VERBS

```
INSPECT identifier-1 TALLYING

  {identifier-2 FOR {CHARACTERS [{BEFORE} INITIAL {identifier-4}] ...              } ...} ...
                    {           [{AFTER }         {literal-2   }]                  }
                    {{ALL    } {identifier-3} [{BEFORE} INITIAL {identifier-4}] ...} ...
                    {{LEADING} {literal-1   } [{AFTER }         {literal-2   }]    }

REPLACING

  {CHARACTERS BY {identifier-5} [{BEFORE} INITIAL {identifier-4}] ...                          }
  {              {literal-3   } [{AFTER }         {literal-2   }]                              } ...
  {{ALL    } {{identifier-3} BY {identifier-5} [{BEFORE} INITIAL {identifier-4}] ...} ...      }
  {{LEADING} {{literal-1   }    {literal-3   } [{AFTER }         {literal-2   }]    }          }
  {{FIRST  }                                                                                   }
```

```
INSPECT identifier-1 CONVERTING {identifier-6} TO {identifier-7}
                                {literal-4   }    {literal-5   }

   [{BEFORE} INITIAL {identifier-4}] ...
   [{AFTER }         {literal-2   }]
```

```
MERGE file-name-1 {ON {ASCENDING } KEY {data-name-1} ...} ...
                      {DESCENDING}

   [COLLATING SEQUENCE IS alphabet-name-1]

   USING file-name-2 {file-name-3} ...

   {OUTPUT PROCEDURE IS procedure-name-1 [{THROUGH} procedure-name-2]}
   {                                     [{THRU   }                 ]}
   {GIVING {file-name-4} ...                                          }
```

```
MOVE {identifier-1} TO {identifier-2} ...
     {literal-1   }
```

```
MOVE {CORRESPONDING} identifier-1 TO identifier-2
     {CORR         }
```

```
MULTIPLY {identifier-1} BY {identifier-2 [ROUNDED]} ...
         {literal-1   }

   [ON SIZE ERROR imperative-statement-1]

   [NOT ON SIZE ERROR imperative-statement-2]

   [END-MULTIPLY]
```

GENERAL FORMAT FOR COBOL VERBS

```
MULTIPLY {identifier-1} BY {identifier-2}
         {literal-1   }    {literal-2   }

    GIVING {identifier-3 [ROUNDED]} ...

    [ON SIZE ERROR imperative-statement-1]

    [NOT ON SIZE ERROR imperative-statement-2]

    [END-MULTIPLY]
```

```
S OPEN  {INPUT {file-name-1  [WITH NO REWIND]} ...  } ...
        {OUTPUT {file-name-2  [WITH NO REWIND]} ... }
        {I-O {file-name-3} ...                      }
        {EXTEND {file-name-4} ...                   }
```

```
RI OPEN {INPUT {file-name-1} ...   } ...
        {OUTPUT {file-name-2} ...  }
        {I-O {file-name-3} ...     }
        {EXTEND {file-name-4} ...  }
```

```
W OPEN {OUTPUT {file-name-1 [WITH NO REWIND]} ... } ...
       {EXTEND {file-name-2} ...                  }
```

```
PERFORM [procedure-name-1 [{THROUGH} procedure-name-2]]
                          [{THRU   }                 ]
    [imperative-statement-1 END-PERFORM]
```

```
PERFORM [procedure-name-1 [{THROUGH} procedure-name-2]]
                          [{THRU   }                 ]
    {identifier-1} TIMES [imperative-statement-1 END-PERFORM]
    {integer-1   }
```

```
PERFORM [procedure-name-1 [{THROUGH} procedure-name-2]]
                          [{THRU   }                 ]
    [WITH TEST {BEFORE}] UNTIL condition-1
    [          {AFTER }]

    [imperative-statement-1 END-PERFORM]
```

GENERAL FORMAT FOR COBOL VERBS

```
PERFORM [procedure-name-1 [{THROUGH} procedure-name-2]]
                           {THRU   }
   [WITH TEST {BEFORE}]
              {AFTER }
   VARYING {identifier-2} FROM {identifier-3}
           {index-name-1}      {index-name-2}
                               {literal-1   }
         BY {identifier-4} UNTIL condition-1
            {literal-2   }
     [AFTER {identifier-5} FROM {identifier-6}
            {literal-3   }      {index-name-4}
                                {literal-3   }
         BY {identifier-7} UNTIL condition-2 ] ...
            {literal-4   }
   [imperative-statement-1 END-PERFORM]

PURGE cd-name-1

SRI READ file-name-1 [NEXT] RECORD [INTO identifier-1]
       [AT END imperative-statement-1]
       [NOT AT END imperative-statement-2]
       [END-READ]

R READ file-name-1 RECORD [INTO identifier-1]
       [INVALID KEY imperative-statement-3]
       [NOT INVALID KEY imperative-statement-4]
       [END-READ]
```

GENERAL FORMAT FOR COBOL VERBS

```
I READ file-name-1 RECORD [INTO identifier-1]

    [KEY IS data-name-1]

    [INVALID KEY imperative-statement-3]

    [NOT INVALID KEY imperative-statement-4]

    [END-READ]


  RECEIVE cd-name-1 {MESSAGE} INTO identifier-1
                    {SEGMENT}

    [NO DATA imperative-statement-1]

    [WITH DATA imperative-statement-2]

    [END-RECEIVE]


  RELEASE record-name-1 [FROM identifier-1]


  RETURN file-name-1 RECORD [INTO identifier-1]

    AT END imperative-statement-1

    [NOT AT END imperative-statement-2]

    [END-RETURN]


S REWRITE record-name-1 [FROM identifier-1]


RI REWRITE record-name-1 [FROM identifier-1]

    [INVALID KEY imperative-statement-1]

    [NOT INVALID KEY imperative-statement-2]

    [END-REWRITE]
```

GENERAL FORMAT FOR COBOL VERBS

```
SEARCH identifier-1 [VARYING {identifier-2 }]
                             {index-name-1 }

    [AT END imperative-statement-1]

    {WHEN condition-1 {imperative-statement-2}} ...
    {                 {NEXT SENTENCE         }}

    [END-SEARCH]


SEARCH ALL identifier-1 [AT END imperative-statement-1]

          {data-name-1 {IS EQUAL TO} {identifier-3             }}
    WHEN  {            {IS =       } {literal-]                }}
          {                          {arithmetic-expression-1  }}
          {condition-name-1                                     }

    [     {data-name-2 {IS EQUAL TO} {identifier-4             }}]
    [ AND {            {IS =       } {literal-2                }}] ...
    [     {                          {arithmetic-expression-2  }}]
    [     {condition-name-2                                     }]

    {imperative-statement-2}
    {NEXT SENTENCE         }

    [END-SEARCH]


SEND cd-name-1 FROM identifier-1


                                      {WITH identifier-2}
SEND cd-name-1  [FROM identifier-1]   {WITH ESI         }
                                      {WITH EMI         }
                                      {WITH EGI         }

    [                    {{identifier-3} [LINE ]}]
    [ {BEFORE} ADVANCING {{integer-1   } [LINES]}]
    [ {AFTER }           {                      }]
    [                    {{mnemonic-name-1}     }]
    [                    {{PAGE           }     }]

    [REPLACING LINE]


SET {index-name-1} ...  TO {index-name-2}
    {identifier-1}         {identifier-2}
                           {integer-1   }
```

GENERAL FORMAT FOR COBOL VERBS

```
SET {index-name-3} ... {UP BY  } {identifier-3}
                       {DOWN BY} {integer-2   }

SET {{mnemonic-name-1} ...  TO {ON }} ...
                               {OFF}

SET {condition-name-1} ...  TO TRUE

SORT file-name-1 {ON {ASCENDING } KEY {data-name-1} ... } ...
                     {DESCENDING}

    [WITH DUPLICATES IN ORDER]

    [COLLATING SEQUENCE IS alphabet-name-1]

    {INPUT PROCEDURE IS procedure-name-1 [{THROUGH} procedure-name-2]}
    {                                     {THRU   }                  }
    {USING {file-name-2} ...                                         }

    {OUTPUT PROCEDURE IS procedure-name-3 [{THROUGH} procedure-name-4]}
    {                                      {THRU   }                  }
    {GIVING {file-name-3} ...                                         }

                      [     {IS EQUAL TO                    }             ]
                      [     {IS =                           }             ]
                      [     {IS GREATER THAN                }             ]
START file-name-1     [ KEY {IS >                           } data-name-1 ]
                      [     {IS NOT LESS THAN               }             ]
                      [     {IS NOT <                       }             ]
                      [     {IS GREATER THAN OR EQUAL TO    }             ]
                      [     {IS >=                          }             ]

    [INVALID KEY imperative-statement-1]

    [NOT INVALID KEY imperative-statement-2]

    [END-START]

STOP {RUN      }
     {literal-1}
```

GENERAL FORMAT FOR COBOL VERBS

```
                                                  {identifier-2}
STRING {{identifier-1} ...  DELIMITED BY {literal-2   }} ...
        {literal-1   }                    {SIZE        }

    INTO identifier-3
    [WITH POINTER identifier-4]
    [ON OVERFLOW imperative-statement-1]
    [NOT ON OVERFLOW imperative-statement-2]
    [END-STRING]
```

```
SUBTRACT {identifier-1} ...  FROM {identifier-3 [ROUNDED]} ...
         {literal-1   }
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-SUBTRACT]
```

```
SUBTRACT {identifier-1} ...  FROM {identifier-2}
         {literal-1   }           {literal-2   }
    GIVING {identifier-3 [ROUNDED]} ...
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-SUBTRACT]
```

```
SUBTRACT {CORRESPONDING} identifier-1 FROM identifier-2 [ROUNDED]
         {CORR         }
    [ON SIZE ERROR imperative-statement-1]
    [NOT ON SIZE ERROR imperative-statement-2]
    [END-SUBTRACT]
```

```
SUPPRESS PRINTING
```

```
TERMINATE {report-name-1} ...
```

GENERAL FORMAT FOR COBOL VERBS

```
UNSTRING identifier-1

    [DELIMITED BY [ALL] {identifier-2} [OR [ALL] {identifier-3}] ... ]
                        {literal-1   }            {literal-2   }

    INTO {identifier-4 [DELIMITER IN identifier-5] [COUNT IN identifier-6]} ...

    [WITH POINTER identifier-7]

    [TALLYING IN identifier-8]

    [ON OVERFLOW imperative-statement-1]

    [NOT ON OVERFLOW imperative-statement-2]

    [END-UNSTRING]
```

```
                                                        ({file-name-1} ...)
USE [GLOBAL] AFTER STANDARD {EXCEPTION} PROCEDURE ON    (INPUT            )
                            {ERROR    }                 (OUTPUT           )
                                                        (I-O              )
                                                        (EXTEND           )
```

```
USE [GLOBAL] BEFORE REPORTING identifier-1
```

```
                        (cd-name-1                             )
                        ([ALL REFERENCES OF] identifier-1      )
USE FOR DEBUGGING ON    (file-name-1                           ) ...
                        (procedure-name-1                      )
                        (ALL PROCEDURES                        )
```

```
S WRITE record-name-1 [FROM identifier-1]

    [                         ({identifier-2} [LINE ])]
    [ {BEFORE} ADVANCING      ({integer-1   } [LINES])]
    [ {AFTER }                (                       )]
    [                         ({mnemonic-name-1}      )]
    [                         ({PAGE           }      )]

    [AT {END-OF-PAGE} imperative-statement-1]
    [   {EOP        }                       ]

    [NOT AT {END-OF-PAGE} imperative-statement-2]
    [       {EOP        }                       ]

    [END-WRITE]
```

```
RI WRITE record-name-1 [FROM identifier-1]

    [INVALID KEY imperative-statement-1]

    [NOT INVALID KEY imperative-statement-2]

    [END-WRITE]
```

GENERAL FORMAT FOR COPY AND REPLACE STATEMENTS

```
COPY text-name-1 [{OF} library-name-1]
                  {IN}

  [            {==pseudo-text-1==}      {==pseudo-text-2==}    ]
  [ REPLACING {{identifier-1      } BY {identifier-2      }} ...]
  [            {literal-1         }     {literal-2         }    ]
  [            {word-1            }     {word-2            }    ]

REPLACE {==pseudo-text-1== BY ==pseudo-text-2==} ...

REPLACE OFF
```

GENERAL FORMAT FOR CONDITIONS

RELATION CONDITION:

```
                           {IS [NOT] GREATER THAN           }
                           {IS [NOT] >                      }
                           {IS [NOT] LESS THAN              }
{identifier-1           }  {IS [NOT] <                      }  {identifier-2           }
{literal-1              }  {IS [NOT] EQUAL TO               }  {literal-2              }
{arithmetic-expression-1}  {IS [NOT] =                      }  {arithmetic-expression-2}
{index-name-1           }  {IS GREATER THAN OR EQUAL TO     }  {index-name-2           }
                           {IS >=                           }
                           {IS LESS THAN OR EQUAL TO        }
                           {IS <=                           }
```

CLASS CONDITION:

```
                         {NUMERIC          }
                         {ALPHABETIC       }
identifier-1 IS [NOT]    {ALPHABETIC-LOWER }
                         {ALPHABETIC-UPPER }
                         {class-name       }
```

CONDITION-NAME CONDITION:

```
condition-name-1
```

SWITCH-STATUS CONDITION:

```
condition-name-1
```

SIGN CONDITION:

```
                                     {POSITIVE}
arithmetic-expression-1 IS [NOT]     {NEGATIVE}
                                     {ZERO    }
```

NEGATED CONDITION:

```
NOT condition-1
```

COMBINED CONDITION:

```
condition-1 {{AND} condition-2} ...
             {OR }
```

ABBREVIATED COMBINED RELATION CONDITION:

```
relation-condition {{AND} [NOT] [relational-operator] object} ...
                    {OR }
```

QUALIFICATION

FORMAT 1:

```
                    ⎧⎧{IN}                ⎫       ⎡{IN}  {file-name}⎤⎫
{data-name-1     }  ⎪⎨{OF} data-name-2    ⎬ ...   ⎣{OF}  {cd-name  }⎦⎪
{condition-name  }  ⎨                                                ⎬
                    ⎪ {IN}  {file-name}                              ⎪
                    ⎩ {OF}  {cd-name  }                              ⎭
```

FORMAT 2:

```
paragraph-name {IN} section-name
               {OF}
```

FORMAT 3:

```
text-name {IN} library-name
          {OF}
```

FORMAT 4:

```
LINAGE-COUNTER {IN} report-name
               {OF}
```

FORMAT 5:

```
{PAGE-COUNTER} {IN} report-name
{LINE-COUNTER} {OF}
```

FORMAT 6:

```
            ⎧{IN} data-name-4  [{IN} report-name]⎫
data-name-3 ⎨{OF}              [{OF}            ]⎬
            ⎪{IN} report-name                    ⎪
            ⎩{OF}                                ⎭
```

MISCELLANEOUS FORMATS

SUBSCRIPTING:

```
{condition-name-1}   ( {integer-1                      }  ... )
{data-name-1     }     {data-name-2 [{±} integer-2]    }
                       {index-name-1 [{±} integer-3]   }
```

REFERENCE MODIFICATION:

```
data-name-1 (leftmost-character-position: [length])
```

IDENTIFIER:

```
data-name-1 [{IN} data-name-2] ... [{IN} {cd-name    }]
            [{OF}            ]     [{OF} {file-name  }]
                                   [     {report-name}]

    [({subscript} ... ) ]  [(leftmost-character-position: [length])]
```

GENERAL FORMAT FOR NESTED SOURCE PROGRAMS

```
IDENTIFICATION DIVISION.

PROGRAM-ID. program-name-1 [IS INITIAL PROGRAM].

[ENVIRONMENT DIVISION. environment-division-content]

[DATA DIVISION. data-division-content]

[PROCEDURE DIVISION. procedure-division-content]

[[nested-source-program] ...

 END PROGRAM program-name-1.]
```

NESTED-SOURCE-PROGRAM:

```
IDENTIFICATION DIVISION.

PROGRAM-ID. program-name-2 [IS {|COMMON|} PROGRAM] .
                                {|INITIAL|}

[ENVIRONMENT DIVISION. environment-division-content]

[DATA DIVISION. data-division-content]

[PROCEDURE DIVISION. procedure-division-content]

[nested-source-program] ...

END PROGRAM program-name-2.
```

GENERAL FORMAT FOR A SEQUENCE OF SOURCE PROGRAMS

```
{IDENTIFICATION DIVISION.

 PROGRAM-ID. program-name-3 [IS INITIAL PROGRAM].

[ENVIRONMENT DIVISION. environment-division-content]

[DATA DIVISION. data-division-content]

[PROCEDURE DIVISION. procedure-division-content]

[nested-source-program] ...

 END PROGRAM program-name-3.} ...

 IDENTIFICATION DIVISION.

 PROGRAM-ID. program-name-4 [IS INITIAL PROGRAM].

[ENVIRONMENT DIVISION. environment-division-content]

[DATA DIVISION. data-division-content]

[PROCEDURE DIVISION. procedure-division-content]

[[nested-source-program] ...

 END PROGRAM program-name-4.]
```

INDEX